

**Domain-Specific Modeling Based Feature-
Oriented Automatic Test Generation Methodology
for Software Product Lines**

**A Thesis Submitted to
the Graduate School of Engineering and Science of
İzmir Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
MASTER OF SCIENCE
in Computer Engineering**

**by
Sercan ŞENSÜLÜN**

May 2019

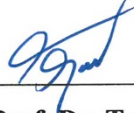
İZMİR

We approve the thesis of **Sercan ŐENSÜLÜN**

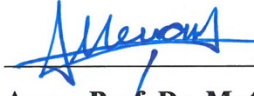
Examining Committee Members:



Assoc. Prof. Dr. Tuđkan TUĐLULAR
Department of Computer Engineering, İzmir Institute of Technology



Assoc. Prof. Dr. Tolga AYAV
Department of Computer Engineering, İzmir Institute of Technology



Assoc. Prof. Dr. Mutlu BEYAZIT
Department of Computer Engineering, Yařar University

31 May 2019



Assoc. Prof. Dr. Tuđkan TUĐLULAR
Supervisor, Department of Computer Engineering
İzmir Institute of Technology



Assoc. Prof. Dr. Tolga AYAV
Head of the Department of
Computer Engineering

Prof. Dr. Aysun SOFUOĐLU
Dean of the Graduate School of
Engineering and Sciences

ACKNOWLEDGMENT

I would like to thank my advisor, who supports me not only academic studies but also industrial practices, Assoc. Prof. Dr. Tuğkan TUĞLULAR. I am sure that we are going to create novel studies in the future like this study.

I also acknowledge to company, which is called as Delta Smart Technologies Inc., and Tolgahan OYSAL who is project manager and founder of Delta Smart Technologies. Thanks to him, proposed solution is proved by a case study.

The research in this thesis is supported by The Scientific and Technological Research Council of Turkey (TUBITAK) under the grant 117E884. The research in this thesis is accepted as a research paper titled as *SPL-AT Gherkin: A Gherkin Extension for Feature Oriented Testing of Software Product Lines* to the *Software Test Automation Workshop 2019 co-located with Computer Software and Application Conference (COMPSAC – STA 2019)*.

Besides, I wish to express my special thanks to my dear friends Gülce ABATAY and Göksu Naz UĞURTAŞ for their technical help and support.

Finally, I am grateful to my mother Veliye ŞENSÜLÜN, my father Sabri ŞENSÜLÜN, my big brother Sedat ŞENSÜLÜN and my grandmother Emine YILMAZ. They always encourage me for my academic researches. I would never complete this study without their spiritual support.

ÖZET

YAZILIM ÜRETİM HATLARI İÇİN ALANA ÖZGÜ MODELLEME TEMELLİ ÖZELLİK ODAKLI OTOMATİK TEST ÜRETME METODOLOJİSİ

Bulut platformları yazılım ürün hatlarına (YÜH) dönüşmektedir. Bu dönüşümle birlikte müşterinin seçtiği özelliklere sahip ürünlerin test edilmesi de büyük bir önem kazanmaktadır. Yazılımın kullanıcının ihtiyaçları doğrultusunda olup olmadığına karar vermek için kabul testleri (KT) kullanılır. Yazılım geliştirme döngüsünde değişen kullanıcı gereksinimleriyle veya müşterinin farklı seçimleriyle birlikte, amaçlanan yazılım ürününün kabul testlerinin geliştirme maliyeti de artmaktadır. Bu çalışma kapsamında özellik bazlı test yaklaşımıyla birlikte SPL-AT Gherkin isimli Gherkin sözdizimine yeni bir uzantı önerilmiştir. Bu önerilen yeni sözdizimi ile birlikte, Test Next Generation (TestNG) çatısını kullanan özgün bir test yöntemleri üreticisi de tasarlanmıştır. Önerilen bu özgün çalışmanın uygulanabilirliği, buton, metin görünümü ve metin düzenleme gibi farklı kullanıcı arayüz bileşenleri olan mobil uygulama platformunda geliştirilen bir uygulama üzerinde denenmiş ve üretilen sonuçlar çıktılarıyla birlikte paylaşılmıştır. Önerilen bu yaklaşım, kullanıcı arayüzüne sahip herhangi bir uygulama üzerinde herhangi bir test çatısıyla birlikte geliştirilmeye açık bir şekilde tasarlanmıştır.

ABSTRACT

DOMAIN-SPECIFIC MODELING BASED FEATURE-ORIENTED AUTOMATIC TEST GENERATION METHODOLOGY FOR SOFTWARE PRODUCT LINES

Cloud platforms are transforming to software product lines (SPLs) and testing of the customer-selected products are becoming increasingly important with this transformation. Acceptance Test (AT) is a testing variety to check acceptability of the software based on user requirements. While user requirements or customer's selection are changing during the development cycle, cost of ATs generation is also increasing. In this study, a feature-oriented testing approach is proposed with a novel extension to Gherkin called SPL-AT Gherkin and a novel automatic test method generation technique that uses Test Next Generation (TestNG) framework. Applicability of the proposed approach is demonstrated with a case study that has different user interface (UI) components such as Page, Button, Text View and Edit Text in mobile application platform. Moreover, results for case study is presented. The proposed approach is open for improvement throughout any application that has UI components such as Web, Mobile with any testing framework.

TABLE OF CONTENTS

LIST OF FIGURES	vii
CHAPTER 1. INTRODUCTION	1
CHAPTER 2. FUNDAMENTALS	3
2.1. Behavior Driven Development.....	3
2.2. Gherkin Syntax.....	3
2.3. Page Object Design Pattern.....	4
2.4. TestNG.....	5
CHAPTER 3. MAPPING RULES	7
3.1. Page Object Design Pattern Part	8
3.1.1. Rule 1	8
3.1.2. Rule 2.....	9
3.1.3. Rule 3.....	11
3.2. TestNG Part.....	11
3.2.1. Rule 4.....	13
3.2.2. Rule 5.....	14
3.2.3. Rule 6.....	15
3.2.4. Rule 7.....	15
3.2.5. Rule 8.....	16
3.2.6. Rule 9.....	19
CHAPTER 4. IMPLEMENTATION of the MAPPING RULE SET.....	26
CHAPTER 5. TOOL SUPPORT	33
5.1. Eclipse.....	33
5.2. Appium.....	33
5.3. Creating Maven Project.....	33
5.4. Adding Libraries to Maven Project.....	34
5.5. Adding Implementation of The Mapping Rule Set to Maven Project.....	34
CHAPTER 6. CASE STUDY.....	36
CHAPTER 7. RELATED WORKS.....	48
7.1. Robot Framework.....	48
7.2. Cucumber.....	50
7.3. Gauge	51
CHAPTER 8. CONCLUSION	53
REFERENCES.....	55

LIST OF FIGURES

<u>Figure</u>	<u>Page</u>
Figure 2.1. Example Test Method	6
Figure 2.2. Example TestNG XML File	6
Figure 2.3. Example Test Method with Priority	6
Figure 3.1. Base Page	8
Figure 3.2. Sample Scenario for Rule 1	9
Figure 3.3. Child of Base Page for Rule 1	9
Figure 3.4. Sample Scenario for Rule 2	10
Figure 3.5. Child of Base Page for Rule 2	10
Figure 3.6. Sample Scenario for Rule 3	12
Figure 3.7. Child of Base Page for Rule 3	12
Figure 3.8. Base TestNG	12
Figure 3.9. Sample Scenario for Rule 4	13
Figure 3.10. Child of Base TestNG for Rule 4	13
Figure 3.11. Sample Scenario for Rule 5	14
Figure 3.12. Child of Base TestNG for Rule 5	14
Figure 3.13. Generated Test Method for Rule 6	16
Figure 3.14. Sample Scenario for Rule 7	16
Figure 3.15. Child of Base TestNG for Rule 7	18
Figure 3.16. Generated TestNG XML File for Rule 8	18
Figure 3.17. Generated Test Method for Rule 8	19
Figure 3.18. Output for Rule 8	17
Figure 3.19. Relations for Given and When Parts	20
Figure 3.20. Relations for Then Part	20
Figure 3.21. Sample Scenario for Rule 9	22
Figure 3.22. Output UML applying Mapping Rule Set	22
Figure 3.23. Generated BasePage for Rule 9	23
Figure 3.24. Generated BaseTestNG for Rule 9	24
Figure 3.25. Generated TestNG XML for Rule 9	25
Figure 4.1. UML for Feature and Scenario Outline	26
Figure 4.2. UML for Step and Example Data Table	27
Figure 4.3. UML for Step and Scenario Outline	27

<u>Figure</u>	<u>Page</u>
Figure 4.4. UML for Example Data Table and Row	28
Figure 4.5. UML for All Gherkin Keywords	28
Figure 4.6. UML with Gherkin Parser	29
Figure 4.7. UML with Tag Finder	30
Figure 4.8. UML with Generators	31
Figure 4.9. UML for Implementation of The Mapping Rule Set.....	32
Figure 5.1. Complete Project Structure.....	35
Figure 6.1. SPL Feature Diagram for KidsBusä.....	37
Figure 6.2. SPL Product Diagram for Gold KidsBusä.....	37
Figure 6.3. Screen for Page Getting SMS Code.....	38
Figure 6.4. Scenario for Getting SMS Code.....	38
Figure 6. 5. Scenario for Getting SMS Code Title	39
Figure 6.6. Screen for Verify SMS Code	40
Figure 6.7. Scenario for Verify SMS Code.....	40
Figure 6.8. Scenario for Count Down Timer Text.....	41
Figure 6.9. Scenario for Send Again Text.....	39
Figure 6.10. Scenario for Title Text.....	41
Figure 6.11. Screen for Create New Password	42
Figure 6.12. Scenario for Create New Password	42
Figure 6.13. Scenario for Create New Password Title	43
Figure 6.14. Main Screen	43
Figure 6.15. Scenario for Main Title	44
Figure 6.16. Login Screen.....	44
Figure 6.17. Scenario for Credential Integrity	45
Figure 6.18. Scenario for Forgot Password Title	45
Figure 6.19. Scenario for Welcome Title	45
Figure 6.20. Test Results.....	46
Figure 6.21. Scenarios' Generation Time in Milliseconds.....	47
Figure 7.1. Robot File with Empty Skeleton.....	49
Figure 7.2. Sample Robot with Appium.....	49
Figure 7.3. Sample SPL-AT Gherkin	50
Figure 7.4. Sample Scenario in Gherkin for Cucumber.....	51

<u>Figure</u>	<u>Page</u>
Figure 7.5. Generated Test Method by Cucumber	51
Figure 7.6. Sample Scenario in Gauge Syntax	52
Figure 7.7. Implementation File for Gauge	52

CHAPTER 1

INTRODUCTION

Nowadays, product customization is becoming more significant with cloud platforms. These platforms serve not only server side but also web or mobile environment solutions. Thanks to these platforms, customers could create unique products for themselves with the selected features. In other words, cloud platforms are new trend of software product lines. While software product lines (SPLs) bring huge advantages in terms of product customization, they also provide some difficulties about reliability of the delivered product. Therefore, quality of the customized product has to be assured before delivering.

Acceptance tests (ATs) are used to ensure that production is ready or not upon to customer requirements. And also, these requirements are changed through the selected features in SPLs. While customer's selection or requirements are changing during the development cycle, cost of ATs generation is also increasing. In this study, we propose a feature-oriented testing approach based on Gherkin but with a novel extension called Software Product Line (SPL) – Acceptance Test (AT) Gherkin. The proposed approach also includes an automatic test method generation technique from SPL-AT Gherkin to concrete acceptance test cases.

The proposed test method generation technique could be changed with any test frameworks. In other words, it is open to change against different environments. The only task is changing implementation of the mapping rules, that is going to be focused in Chapter 3, with intended framework. Addition to them, acceptance test driven development can be followed with the proposed approach.

In SPLs, while some features are defined as default such as *demonstrating list of items* or *current status of logged user* for each application, other features could be added based on customer's requirements such as *tracking status of items* or *adding item to stock*. In other words, different variation of the applications could be generated easily in SPLs based on customers' selected features. With this variety, each application, which is generated in the SPL, should be tested before delivering to the customer. One of the

motivations behind the proposed test method generation technique is handling this challenge. Therefore, this method could be applied for any SPL which generates application that has User Interface (UI) components. To prove this idea, KidsBusÖ system, which was not designed as SPL but suitable to apply this approach, is used in Case Study part.

The study is organized generally as follows. After explaining fundamentals in Chapter 2, proposed approach is focused in detail under Chapter 3. Then implementation of proposed approach is explained with Unified Modeling Language (UML) in Chapter 4. After explaining proposed approach and implementation clearly, development environment preparation is demonstrated step by step in Chapter 5 that is called as Tool Support. Then, the approach is applied to an application, which called as KidsBusä School Security, in Chapter 6. After that, in Chapter 7, related works are handled with similarities and dissimilarities to proposed approach. In Conclusion Chapter, the study is handled in broad perspective.

CHAPTER 2

FUNDAMENTALS

2.1. Behavior Driven Development

In agile development technique, information gap between stakeholders and developers is tried to be reduced. User stories are used to reduce this gap. Each scenario, that is written in natural language by stakeholders, should correspond to a piece of code. Acceptance tests are used to ensure this match. Test Driven Development (TDD) offers that writing Acceptance Tests (ATs) first then writing the code, which is evaluated by the tests. When the code tested by all ATs, software could be assumed as complete with respect to acceptance criteria. While applying TDD, Dan North encountered some misunderstanding between analysts, developers and business people. To reduce this mismatch, he proposed Behavior Driven Development. In BDD [1], the scenarios are written in spoken language, e.g. English, Turkish. Thanks to these scenarios, acceptance criteria are more understandable by all team members, e.g. Product Owners, Testers, UX Designer, Programmers. There are various structures to write a scenario, e.g. Given When Then [2]. In other words, they are structured documentation waiting to be processed in different purposes, e.g. generating ATs. Martin Fowler also establishes connection between Given When Then and Gerard Meszaros' three phases of Four-Phase Test [3] which are Setup, Exercise and Verify respectively [2]. In Chapter 3, this is going to be focused in deeper in terms of acceptance test generation for applications which have User Interface (UI) components such as buttons, text views or editable views.

2.2. Gherkin Syntax

Gherkin [5] is a domain specific language to create project documentation and automated tests. It provides the behavior definitions of the intended software not only to product owners and business analysts but also to developers and testers. In other words, it is a well-known language, which is understandable by any teams with +70 spoken

languages support. Gherkin is a line-oriented language in terms of structure and each line has to be divided by the Gherkin keyword except feature and scenario descriptions. Some of the Gherkin keywords, which are Scenario Outline, Given, When, And, Then, Examples, are going to be handled in describing Software Product Line-Acceptance Test (SPL-AT) Gherkin.

Scenario Outline is one of the keywords in Gherkin. Thanks to its structure, different scenarios could be executed in same scenario skeleton. In Scenario Outline, parameters, that are different for each scenario, are defined with in < > characters, i.e. <parameter>. Furthermore, they should be included *Examples* data table [5]. In *Examples*, the first row must include all described parameters in to the *Scenario Outline*. Variation of the parameters are defined in the following rows respect to order of the first row.

2.3. Page Object Design Pattern

PageObject design pattern was introduced for web pages to hide User Interface details from client. It is a basic encapsulation mechanism because it finds to UI components such as Header or Paragraph tags in HTML page and manipulates it without any technical details, i.e., Web Driver. While writing test against any web page, it is suitable to manipulate UI components. Despite Martin Fowler explains this pattern for web pages, he claims that it could be applied to any User Interface technology [4]. According to his opinion, this pattern is going to be evolved to the Mobile Application Testing domain in the following parts. For instance, you have a mobile application that includes one page which is called as Login Page. It contains one editable field which is called as EditText in Android or UITextField in iOS and one button to validate written text in the editable component. While writing test cases to this page, one of the test frameworks should use to manipulate it, i.e., Appium. Accessor methods, e.g., getText(), setText(...) could developed for editable field, and also button could represented by action oriented methods, e.g., clickButton(). Appium API (Application Programming Interface) methods as technical details are hidden behind these methods. Thanks to this encapsulation, test methods, which are generated by test cases, could be improved with accessor and action-oriented methods without knowledge of Appium API.

2.4. TestNG

TestNG (Test Next Generation) is a testing framework, that is inspired by JUnit, for Java developers [6]. It is suitable to write unit, functional, end-to-end, integration etc. tests. It also suitable for test automation frameworks, e.g. Selenium, Appium. It could be plugged some integrated development environment such as Eclipse, IntelliJ IDEA in to use. It supports some strong features such as data-driven testing, parametrized testing and flexible test configuration. Test methods could take one or more parameters. With this feature, different parameters could be passed to same test method in different scenarios. Parameters could set two different ways, with testing.xml or programmatically. During the research, testing.xml is going to be used. Imagine that, you have a java method that multiples given integer parameter with 2 and returns the result. To test a method with three different parameters which are -1, 0, +1, testing.xml should generated as Figure 2.2. When the test method executed in Figure 2.1, these three test cases are going to be executed with only one test method. Another important feature for the research is priority. If order of the test cases execution is critical, priority should use with *@Test* tag. Priority is represented by integers and lower value is executed first. In Figure 2.3, *test_method_first* always executes before the *test_method_second*.

```

@Parameters({"param"})
@Test
public void Test_Method (String param)
{
//send param to the multiplier here.
}

```

Figure 2.1. Example Test Method

```

<suite name="Suite">
    <test name="multiplewithminusone">
        <parameter
name="param" value="-1"></parameter>
    </test>
    <test name=" multiplewithzero">
        <parameter
name="param" value="0"></parameter>
    </test>
    <test name=" multiplewithplusone">
        <parameter
name="param" value="+1"></parameter>
    </classes>
    </test>
    <...>

```

Figure 2.2. Example TestNG XML File

```

//rest of the test class...
@Test(priority = 0)
public void test_method_first()
{
    //execute firstly.
}

@Test(priority = 1)
public void test_method_second()
{
    //execute secondly.
}
//rest of the test class...

```

Figure 2.3. Example Test Method with Priority

CHAPTER 3

MAPPING RULES

Gherkin is efficient language to write User Scenarios. However, it is not sufficient to generate Acceptance Tests for Mobile Applications. Main purpose of Mapping Rules is transition between User Scenarios and automatically generated Acceptance Test Project. If User Interfaces and their behavior are defined in User Scenarios, transition would be easy and tag structures are generated to achieve this convenience.

There are two different tag structures, which are *address sign (@)* and *dollars (\$)*. They are added on Gherkin to write convertible scenarios for executable Acceptance Tests. While @ tag is used to define User Interface Components such as Edit Text, Button, Text View etc., \$ tag is used to define their behaviors. Thanks to usage of them, user scenario writers could refer application components. @ tag has four different sub-tags which are @PAGE, @EDIT_TEXT, @BUTTON, @TEXT_VIEW and to define their action \$ENTERED, \$CLICKED, \$SHOWN, \$ENABLED, \$DISABLED, \$OPENED tags are generated. After usage of the @ tag, identifier of the UI component should be indicated. The identifier should be found in the Mobile Application project. Relation between @ and \$ are going to be considered in the following sections. While explaining mapping rules in the following parts, Page Object Design Pattern and some of TestNG framework are going to be focused.

There are eleven rules to generate automatically Acceptance Test Project from feature files which are written in SPL-AT Gherkin. The rules are divided in two groups. In the first group, there are three rules, which are related with Page Object design pattern that is briefly mentioned above. And, second group has eight rules that are going to be processed with TestNG framework. After explaining them one by one, case study, which is a commercial mobile application developed by Delta Smart Technologies Inc., is going to explained in the following chapter.

3.1. Page Object Design Pattern Part

Before explaining each rule, some assumptions should be mentioned. In our proposed solution, there is a *Base Page* class, Figure 3.1, and it manages Appium API methods. It also has five methods which are called as *click()*, *setText()* and *getText()*, *isEnabled()* and *isShown()*. *click()* method is responsible to clickable UI components, i.e. button. Addition to this, *setText()* is about editable UI components, i.e., EditText or Text Area. And also, *getText()* helps us to get text which are represented on the UI components such as TextView. *isEnabled()* and *isShown()* with Boolean return type help us to be ensure about visibility of any UI components. All of them take identifier (String) parameter to ensure which UI component is going to be referred on application under test. In the future, if another test automation framework is considered, implementation of these methods is going to be changed with chosen framework API. In other words, the solution is open to change with other frameworks. Finally, before introducing to the rules, each future file which is written in SPL-AT Gherkin has to include only one *Scenario Outline* that was mentioned in previous chapters.

BasePage
- driver: AndroidDriver
- wait: WebDriverWait
click(String identifier): void
setText(String identifier, String text): void
getText(String identifier): String
isEnabled(String identifier): boolean
isShown(String identifier): boolean

Figure 3.1. Base Page

3.1.1. Rule 1

The first rule is related with child classes of the *Base Page* class. *Scenario Outline*, which is in the feature file, has to includes only one *@PAGE* tag with same identifier in *Given* and *When* parts. If only one page on mobile application is going to be tested, *@PAGE* sub-tag with another identifier only could be existed into *Then* part of

the SPL-AT Gherkin. If `@PAGE` tag is detected with identifier, child of the *Base Page* should be created into Acceptance Test Project with *identifierPage* name. For instance, in Figure 3.2, there is one `@PAGE` tag with *this_is_identifier* identifier. According to the rule, *ThisIsIdentifierPage* class that is the child of the *Base Page* class, in Figure 3.3, should be created into the Project.

`@tag`

Feature: This is the title of the Feature

Scenario Outline: This is the title of the Scenario Outline

Given `@PAGE` *this_is_identifier* is opened

When ...

Then ...

Figure 3.2. Sample Scenario for Rule 1

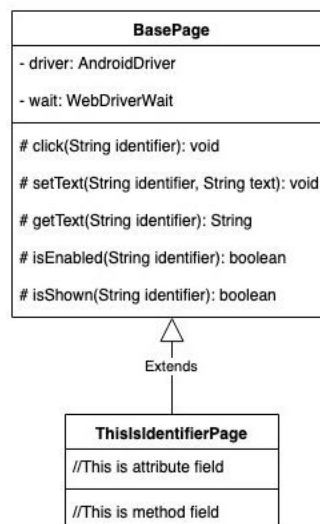


Figure 3.3. Child of Base Page for Rule 1

3.1.2. Rule 2

In second rule, inside of child class that is mentioned in Rule 1 is going to be processed. `@EDIT_TEXT` tag could be existed in *Given* and *When* parts in *Scenario Outline*. Addition to this, *delimited parameter* in Gherkin [5] has to be within the same part. When it is detected with delimited parameter, there should be a method into the child class to set any text to mentioned UI component via `@EDIT_TEXT` tag. Moreover,

method of the *Base Page*, which is *setText(String identifier, String text)*, should exist inside of the generated method with the given identifier from SPL-AT Gherkin. For instance, in Figure 3.4, `@EDIT_TEXT` tag exists in *When* part with *this_is_edit_text_identifier* identifier. There is also *delimited parameter*, that is shown with `<>` special characters, in *When* part and value of the parameter is defined on *Example* data table [5]. When the rule is applied, UML is going to be changed as Figure 3.5. The point is that UML design is generated automatically so that the rule could be applied for any Scenario Outline.

```

@tag
Feature: This is the title of the Feature

Scenario Outline: This is the title of the Scenario
Outline Given @PAGE this_is_identifier is opened
When
<delimited_parameter> is entered on
@EDIT_TEXT this_is_edit_text_identifier
Then ...

Examples:
| delimited_parameter |
| "this_is_value_for_edit_text" |

```

Figure 3.4. Sample Scenario for Rule 2

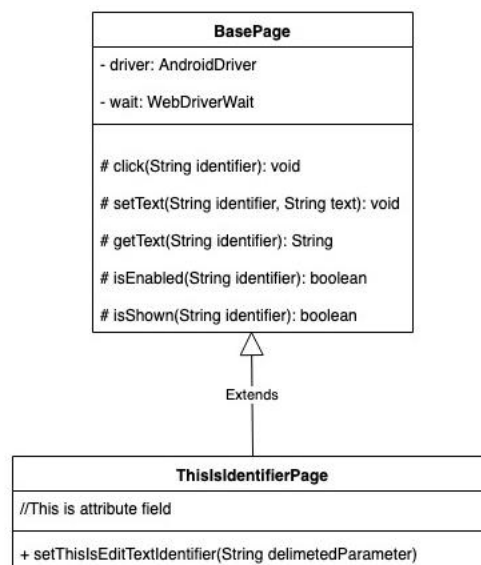


Figure 3.5. Child of Base Page for Rule 2

3.1.3. Rule 3

`@BUTTON` tag is going to be focused in Rule 3. This tag has to be in *Given* and *When* parts for the Rule 3. In fact, pattern of this rule resembles to Rule 2. Because, it is going to generate a filled method to inside of the child class, which is generated in Rule 1, for the clickable UI component, i.e., Button. When the tag is detected with the identifier, action-oriented method has to be created inside of the child class. Then, *click(String identifier)* method, that was implemented into the *Base Page* super class, should be put into this method. For instance, in Figure 3.6, the sub-tag is found in *When* part with *this_is_button_identifier* identifier. As a result, UML design, shown in Figure 3.7, is generated automatically.

3.2. TestNG Part

In the second group, eight rules that are related with the TestNG framework are going to be analyzed. Test classes that collaborate with the Page classes, are going to be generated based on the feature file that was written in SPL-AT Gherkin. During the generation, parameterized test (`@Parameters`) and test prioritization (`@Priority`) topics are going to be used in TestNG framework. Before explaining these rules, *Base TestNG* class has to be focused. Like *Base Page* class, mentioned in section 3.1, it should also be included for each test project. So that, it has to be generated before applying these eight rules. In this class, there are two methods which are called as *setUp()* and *tearDown()*. First method, *setUp()*, is tagged with `@BeforeClass` TestNG annotation. It runs before the first test method that takes part in the same test class. According to this feature, all Appium driver configurations are set in this method. For instance, Unique Device Identifier (UDID) [7] of the mobile device under test or package name of the application under test have to set in it. When any configuration values are changed, this part of the *Base TestNG* class is going to be handled. Moreover, the second method, *tearDown()*, has `@AfterClass` TestNG annotation. Unlike *setUp()* method, it runs after all test methods that takes part in the same test class. So that, some rollback operations such as closing Appium driver are managed in this method. In the following rule parts, this class is going to be extended by the other test classes.

@tag

Feature: This is the title of the Feature

Scenario Outline: This is the title of the Scenario

Outline Given @PAGE *this_is_identifier* is opened

When

<delimited_parameter> is entered on

@EDIT_TEXT *this_is_edit_text_identifier*

And @BUTTON *this_is_button_identifier* is

pressed Then ...

Examples:

```
| delimited_parameter |  
| "this_is_value_for_edit_text" |
```

Figure 3.6. Sample Scenario for Rule 3

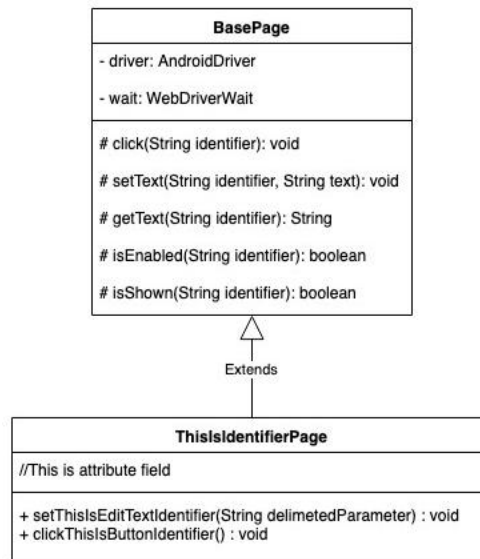


Figure 3.7. Child of Base Page for Rule 3

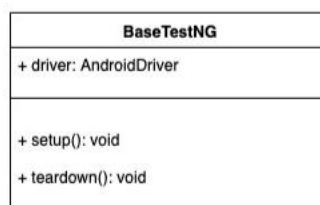


Figure 3.8. Base TestNG

3.2.1. Rule 4

At the end of the Part 3.1, “each feature file, which is written in SPL-AT Gherkin has to include only one *Scenario Outline*” assumption is mentioned. So that, number of the feature files equals to number of the scenarios. Moreover, scenario could be detected easily when feature file is detected. Definition of the rule is that every scenario, in other words, every *Scenario Outline* is a sub-class of the *BaseTestNG*. For instance, in Figure 3.9, title of the *Scenario Outline* which, called as *This is the title of the Scenario Outline*, is going to be converted to name of the class. When the definition is applied, Figure 3.10 is going to be generated.

```
@tag
Feature: This is the title of the Feature

Scenario Outline: This is the title of the Scenario
Outline Given ...
When ...
Then ...
Examples:
| ... |
| ... |
```

Figure 3.9. Sample Scenario for Rule 4

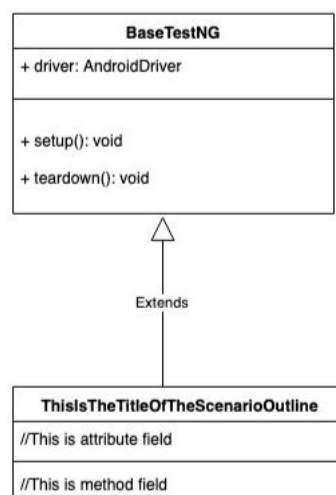


Figure 3.10. Child of Base TestNG for Rule 4

3.2.2. Rule 5

When any *Scenario Outline* is analyzed, three base keywords, which are *Given*, *When*, *Then* are noticed. Moreover, each keyword describes itself with one sentence. Rule 5 claims that each keyword is going to be converted to a TestNG test method with *@Test* annotation into the child of the *BaseTestNG* class that was described in Rule 4. So that, number of the test methods are going to be equal to number of the keywords that exist into these three base keywords. When the rule is applied on Figure 3.11, Figure 3.12 is going to be generated.

@tag
Feature: This is the title of the Feature

Scenario Outline: This is the title of the Scenario Outline
Given this is the Given sentence
When this is the When sentence
Then this is the Then sentence
Examples:
| ... |
| ... |

Figure 3.11. Sample Scenario for Rule 5

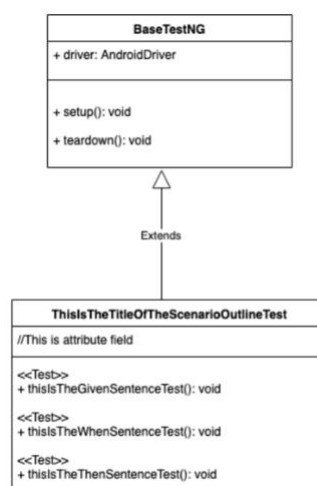


Figure 3.12. Child of Base TestNG for Rule 5

3.2.3. Rule 6

There is a hierarchy between *Given*, *When* and *Then* keywords in terms of the execution order. *Given* keyword is described as initialization part of the scenario such as opening the application page. Addition to this, *When* keyword has some event based operations, e.g. click button, set username in to text field. And, in *Then* keyword, some assertion operations are found, e.g. page is opened or button is disabled. In summary, test methods, which were generated in Rule 5 based on *Given-When-Then* template, have to be executed in a sequence. So that, sorting of the methods has to be *Given*, *When*, *Then* in respect to execution order. *@priority* TestNG, shown in Figure 3.13, annotation is going to be used to implement this order.

3.2.4. Rule 7

Parameterized tests are important topic in automation testing. Different test cases could be handled clearly with it. Moreover, same test method could be executed with different test inputs. When Rule 6 considered, there are three test methods which are *thisIsTheGivenSentence*, *thisIsTheWhenSentenceTest*, *thisIsTheThenSentenceTest* in *ThisIsTheTitleOfTheScenarioOutlineTest* class. And, one of the aimed solution is that execute many test cases with these methods. To achieve this, *Examples* Table, which is defined in feature file, is going to be converted to *@Parameters* TestNG annotation. In *Examples* Table, header row should be represented into *Scenario Outline*. And, other rows represent value of each cell of header row. For instance, in Figure 3.14, there are two different delimited parameters which are *delimited_parameter_1* and *delimited_parameter_2* in *Scenario Outline*. And also, value of these parameters appears in second and third row of *Examples* tables, i.e. *this_is_value_for_param_1*, *this_is_value_for_param_2* etc. The rule argues that when any delimited parameter detected on *Scenario Outline*, it is going to be converted to parameter of the test method. For instance, *delimited_parameter_1* is going to be defined as parameter to *thisIsTheGivenSentenceTest* test method.


```

public class ThisIsTheTitleOfTheScenarioOutlineTest extends
BaseTestNG{

    //This is attribute field

    @Test(priority = 0)
    public void thisIsTheGivenSentenceTest (){
        //firstly executed
    }

    @Test(priority = 1)
    public void thisIsTheWhenSentenceTest (){
        //secondly executed
    }

    @Test(priority = 2)
    public void thisIsTheThenSentenceTest (){
        //thirdly executed
    }
}

```

Figure 3.13. Generated Test Method for Rule 6

```

@tag
Feature: This is the title of the Feature

Scenario Outline: This is the title of the Scenario Outline
Given this is the Given sentence
<delimited_parameter_1> When this is the When sentence
Then this is the Then <delimited_parameter_2> sentence
Examples:
| delimited_parameter_1 | delimited_parameter_2 |
| "this_is_value_for_param_1"|"this_is_value_for_param_2"
| "this_is_a_value_for_param_1"|"this_is_a_value_for_param_2"|

```

Figure 3.14. Sample Scenario for Rule 7

3.2.5. Rule 8

After setting parameter annotations in the test class, which was called as *ThisIsTheTitleOfTheScenarioOutlineTest* in previous rule parts, values of these parameters should be passed to the test methods, i.e. *thisIsTheGivenSentenceTest*,

thisIsTheThenSentenceTest. Passing parameters values through *testng.xml* is one of the passing manner in TestNG framework [8]. *Testng.xml* file is a configuration file to manage test suite and its parameters in any test project. There are many different xml tags such as `<test>`, `<parameters>` etc. in *testng.xml* configuration file. In Rule 8, `<test>`, `<parameters>`, `<classes>` and `<class>` are going to be focused. When number of rows are detected, as the first step, on *Examples Tables*, `<test>` tag is going to be generated for each of them with *name* attribute. This *name* attribute should be unique to identify test case, that's why it was considered as GUID [9] string. Then, as the second step, `<parameter>` tags with *name* and *value* attributes will be generated for each parameter, that was considered in Rule 7, into the `<test>` tag. After that, as the third step, `<class>` tag with *name* attribute will also be generated in to the `<test>` tag. These three steps are run for every rows of *Examples Table* except the header row. For instance, in Figure 3.14, there are two rows that are in Rule 8 scope. When the first and second steps are applied, two `<test>` tags with different *name* attributes will be generated with two `<parameter>` tags into the *testng.xml* file. Then, according to the third step, one `<class>` tag will be generated in to the each `<test>` tag. The key point in Rule 8, each `<test>` tag should be assigned with different names and also same `<class>` name. As a result, Figure 3.17, that describes *testng.xml* file, will be generated with the Rule 8. In Figure 3.18, test class, that was called as *ThisIsTheTitleOfTheScenarioOutlineTest*, was changed with some line of codes to be more understandable about Rule 8. When the test class is run with the *testng.xml* file, test outputs will be generated as Figure 3.15. Thanks to Rule 8, many different test cases with the different parameter values could run with only these three test methods.

```

|[RemoteTestNG] detected TestNG version 6.14.3
thisIsTheGivenSentence param is this_is_value_for_param_1
thisIsTheWhenSentence
thisIsTheThenSentence param is this_is_value_for_param_2
thisIsTheGivenSentence param is this_is_a_value_for_param_1
thisIsTheWhenSentence
thisIsTheThenSentence param is this_is_a_value_for_param_2

```

Figure 3.15. Output for Rule 8

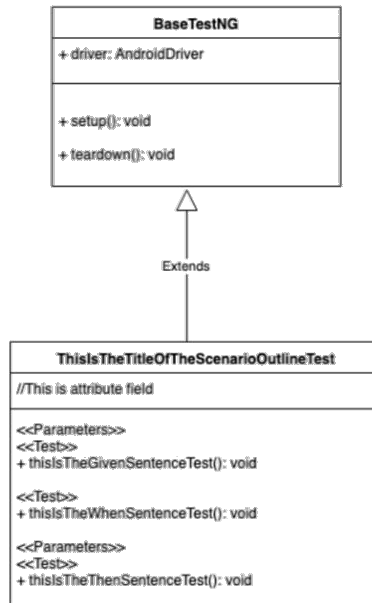


Figure 3.16. Child of Base TestNG for Rule

```

<suite name="Suite">
<test name="74129e81-7ce2-458b-8683-0a235978dc98"> <parameter
    name="delimited_parameter_1"
    value="this_is_value_for_param_1">
</parameter>
<parameter
    name="delimited_parameter_2"
    value="this_is_value_for_param_2">
</parameter>
<classes>
<class name="Tests.ThisIsTheTitleOfTheScenarioOutline">
</class>
</classes>
</test>
<test name="7f935cad-8d28-4dc4-8fc0-725286b83f87">
<parameter
    name="delimited_parameter_1"
    value="this_is_a_value_for_param_1">
</parameter>
<parameter
    name="delimited_parameter_2"
    value="this_is_a_value_for_param_2">
</parameter>
...
  
```

Figure 3.17. Generated TestNG XML File for Rule 8

```

public class ThisIsTheTitleOfTheScenarioOutline extends
BaseTestNG{

    //This is attribute field

    @Parameters({"delimited_parameter_1"})
    @Test(priority = 0)
    public void thisIsTheGivenSentence(String
param){ //firstly executed
    System.out.println(
    "thisIsTheGivenSentence " +
    " param is " + param);
    }

    @Test(priority = 1)
    public void thisIsTheWhenSentence(){
    //secondly executed
    System.out.println("thisIsTheWhenSentence");
    }

    @Parameters({"delimited_parameter_2"})
    @Test(priority = 2)
    public void thisIsTheThenSentence(String param){
    //thirdly executed
    System.out.println(
    "thisIsTheThenSentence " + " param is " + param);
    }
}

```

Figure 3.18. Generated Test Method for Rule 8

3.2.6. Rule 9

Up to this rule, PageObject design pattern and TestNG parts are covered. However, these two main concepts are not connected to each other. In other words, inside of the test methods generated by Rule 5 are not filled with the proper line of codes. Not only content of the test methods but also another tag structure, which was called as *dollars* (\$), and relation between *dollars* (\$) and *address sign* (@) will be covered in this section.

When the rule set, except Rule 8, is review, somehow the PageObject design pattern methods, which were covered in Rule 2 and Rule 3, and the test methods, which were focused on Rule 5, should work with together. To achieve this goal, \$ tag structure

and its relation with @ are thought. It has different adjective keywords such as \$ENTERED, \$OPENED, \$CLICKED, \$ENABLED, \$DISABLED. In relation perspective, these \$ tags should be used with @ in Given, When and Then parts, which are covered in SPL-AT Gherkin. In Given and When parts, while @PAGE should be used only with \$OPENED, \$CLIKED should take part with @BUTTON. And also, \$ENTERED ought to be used with @EDIT_TEXT. In Figure 3.19, these correlations are represented clearly with their identifier. In Then part, @PAGE should be use with @MOVED, and also, @EDIT_TEXT and @BUTTON ought to take part with \$ENABLED or \$DISABLED, see also Figure 3.20. Briefly, these correlations should be existed in Scenario Outline to fill inside of the test methods with the correct lines of codes.

Relations for Given and When Parts

Identifier	@ Tags	\$ Tags
C1	@PAGE	\$OPENED
C2	@BUTTON	\$CLICKED
C3	@EDIT_TEXT	\$ENTERED

Figure 3.19. Relations for Given and When Parts

Relations for Then Part

Identifier	@ Tags	\$ Tags
C4	@PAGE	\$OPENED
C5	@BUTTON	\$ENABLED, \$DISABLED
C6	@EDIT_TEXT	\$ENABLED, \$DISABLED
C7	@TEXT_VIEW	\$SHOWN

Figure 3.20. Relations for Then Part

Until this part of the Rule 9, correlations between @ and \$ were mentioned with their identifier. And now, converting these relations to lines of codes, which are going to be set to inside of the test methods, will be focused. While focusing on any correlation, it will be demonstrated as *C identifier*. For instance, *C1* will be used to

refers relation between *@PAGE* and *\$OPENED* in *Given* and *Then* parts. If *C3* is detected on *Scenario Outline*, then the PageObject design pattern method, that was generated on Rule 2, will be written inside of the *Given* or *When* test method. And also, if *C2* is noticed, then the test method will be implemented with the method which was generated on Rule 3. While code generation is understandable easily for *Given* and *When* parts, on the other hand, it is not quite understandable for *Then* part.

In *Then* part, *\$OPENED*, *\$ENABLED*, *\$DISABLED* and *\$SHOWN* tags are available for *@PAGE*, *@BUTTON*, *@EDIT_TEXT* and *@TEXT_VIEW*. If *C4* is detected with the *<delimited>* parameter in *Then* part, it is converted to *assertEquals(String actual, String expected)* line of code into the *Then* test method, which was focused on Rule 5. The critical part for this line of code is *actual* and *expected* values, because it will be assertion part for the test scenario. In other words, it will decide that the test is fail or not. To determine *actual* part, Appium driver method, which is as called *currentActivity()* [10], will be used. In other words, actual part will be assigned to returned value of this method. On the other hand, expected value could be assigned easily with the value of the delimited parameter, which was occurred in Examples Table in SPL-AT Gherkin, see also Rule 7. If one of the correlations, *C5* or *C6* or *C7*, is detected on *Then* part, *assertTrue(boolean condition)* methods will take part into the *Then* test method. The task for *C5*, *C6* and *C7*, that should be considered, is how we decide value of the *condition* parameter for *assertTrue(Boolean condition)* method. To implement this task, *isEnabled()* [11] Appium driver method will be used. It determines that element, which could be edit text or button for our domain, is currently enabled or not. In other words, value of this method will be assigned to condition variable, which is passed as parameter to *assertTrue(...)* method.

To sum up, implementation of the three test methods, which were focused on Rule 5, was studied in Rule 9. Figure 3.21 is written to be more understandable for not only this rule but also previous rules. If the rule set that is occurred from Rule 1 to Rule 9 is applied to *Scenario Outline* in Figure 3.21, UML diagram, in Figure 3.22, and implementation of the classes, in Figure 3.23 and Figure 3.24, and also *testng.xml* file, in Figure 3.25, is generated automatically. In other words, *Scenario Outline* that was written with SPL-AT Gherkin is converted to implemented Mobile Application Test Project.

@tag

Feature: This is the title of the Feature

Scenario Outline: This is the title of the Scenario

Outline Given @PAGE *this_is_identifier* is \$OPENED

When <parameter_for_edit_text> is \$ENTERED on

@EDIT_TEXT *this_is_edit_text_identifier*

And @BUTTON *this_is_button_identifier* is

\$CLICKED Then Application is

\$OPENED @PAGE <parameter_for_page>

Examples:

```
| parameter_for_edit_text | parameter_for_page |  
| "this_is_value_1_for_edit_text" | "this_is_value_1_for_page" |  
"this_is_value_2_for_edit_text" | "this_is_value_2_for_page" |
```

Figure 3.21. Sample Scenario for Rule 9

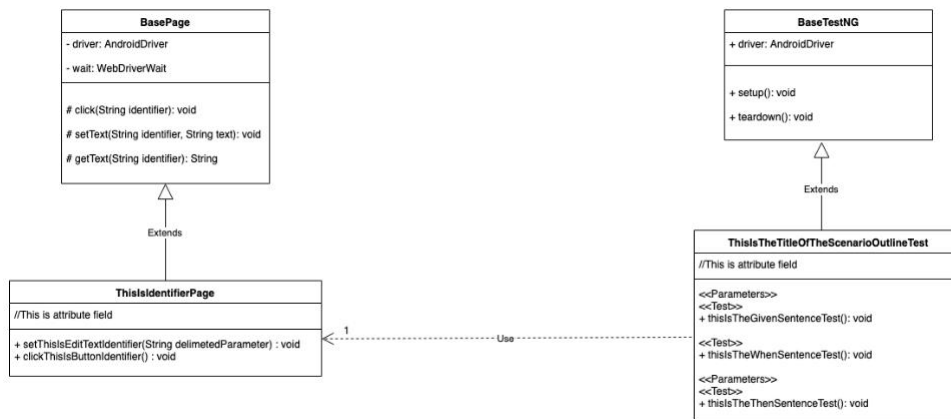


Figure 3.22. Output UML applying Mapping Rule Set

```

public class ThisIsIdentifierPage extends BasePage{
    public ThisIsIdentifierPage(AndroidDriver driver,
WebDriverWait wait)
    {
        super(driver, wait);
        // ...
    }

    /*
    * This method is Auto-generated by the rule 2.
    * */
    public void setThisIsEditTextIdentifier(String
delimetedParameter){

        super.setText(
"thisIsEditTextIdentifier", delimetedParameter);
    }

    /*
    * This method is Auto-generated by the rule 3.
    * */
    public void clickThisIsButtonIdentifier(){
        super.click("thisIsButtonIdentifier");
    }
}

```

Figure 3.23. Generated BasePage for Rule 9


```

public class ThisIsTheTitleOfTheScenarioOutline extends
BaseTestNG{

    //This is attribute field
    ThisIsIdentifierPage page = new ThisIsIdentifierPage(driver,
wait);

    @Test(priority = 0)
public void
Given_PAGE_this_is_identifier_is_OPENED(String param){
    //firstly executed
    //start appium here.
}

    @Parameters({"parameter_for_edit_text"})
    @Test(priority = 1)
public void
When_parameter_for_edit_text_is_ENTERED_on_EDIT_TEXT_this_is
_edit_text_identifier(String param){
    //secondly executed
    page.setThisIsEditTextIdentifier(param);
}

    @Test(priority = 2)
public void
And_BUTTON_this_is_button_identifier_is_PRESSED(){
    //thirdly executed
    page.clickThisIsButtonIdentifier();
}

    @Parameters({"parameter_for_page"})
    @Test(priority = 3)
public void
Then_Application_MOVED_PAGE_parameter_for_page(String param){
    //fourthly executed
    assertEquals(param, ((AndroidDriver<MobileElement>)
driver).currentActivity());
}
}

```

Figure 3.24. Generated BaseTestNG for Rule 9

```

<suite name="Suite">
  <test name="74129e81-7ce2-458b-8683-
    0a235978dc98"> <parameter
      name="parameter_for_edit_text" value
        ="this_is_value_1_for_page"> </parameter>
    <parameter name="parameter_for_page"
      value="this_is_value_1_for_page">
      </parameter>
      <classes>
    <class name="Tests.ThisIsTheTitleOfTheScenarioOutline">
    </class>
      </classes>
    </test>
    <test name="7f935cad-8d28-4dc4-8fc0-
      725286b83f87"> <parameter
        name="parameter_for_edit_text" value
          ="this_is_value_2_for_page"> </parameter>
    <parameter name="parameter_for_page"
      value="this_is_value_2_for_page">
      </parameter>
      <classes>
    <class name="Tests.ThisIsTheTitleOfTheScenarioOutline">
    </class>
      </classes>
    </test>
  </suite>

```

Figure 3.25. Generated TestNG XML for Rule 9

CHAPTER 4

IMPLEMENTATION of the MAPPING RULE SET

In previous chapter, mapping rule set, that converts feature files, which are written in SPL-AT Gherkin, to Mobile Application Test Project, which has two different concepts as PageObject design pattern and TestNG parts, are explained step by step. Object Oriented Programming (OOP) paradigm will be used to implement these rules to Java language. Moreover, in design part, Unified Modeling Language (UML) will assist to understand concepts of implementation part.

Firstly, all Gherkin keywords, which are *Scenario Outline*, *Given*, *When*, *Then*, *And*, *Example*, *Feature* are translated to objects. In terms of OOP, when any feature file is analyzed, the opinion that every *Feature* could has one or many *Scenario Outlines* is realized. However, in this study, it should only one *Scenario Outline*. With these ideas, *Feature* class is designed to have many *Scenario Outline* objects to support future works, in Figure 4.1.

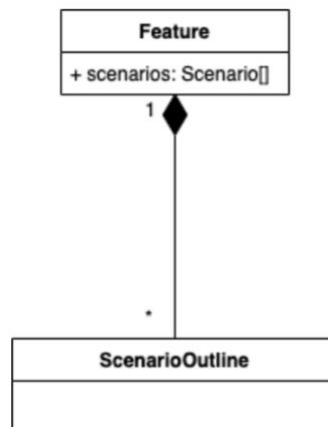


Figure 4.1. UML for Feature and Scenario Outline

After designing Feature object, *Scenario Outline* was needed to extend Gherkin keywords design part. In Gherkin, A *Scenario Outline* has not only many Steps, which are *Given*, *When*, *Then*, *And* but also one Examples table [5]. According to this argument, two different classes, which are *Step* and *Example Data Table*, are designed

as list attribute into the *Scenario Outline* class to support future works as previous. And also, in Figure 4.2, *name* and *id* attributes are considered to identify *Scenario Outline*. In the following part of the implementation, these identifiers will be used as test methods and classes name.

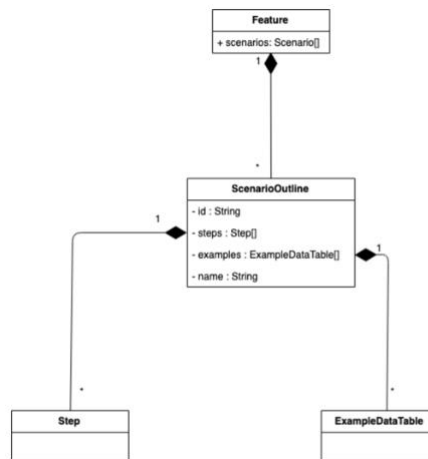


Figure 4.2. UML for Step and Example Data Table

Every line that is divided by *Given*, *When*, *Then* or *And* in *Scenario Outline* is a *Step* in Gherkin [5]. To identify each *Step*, *name*, *keyword* and *line* attributes are put as string, string, integer types respectively into *Step* class. While *keyword* is identifier for *Step*, e.g. *Given*, *name* has whole sentence after the *keyword*. And also, *line* refers to line number of the *Step*. *Name* attribute will be used as the test method name that was focused on Rule 5.

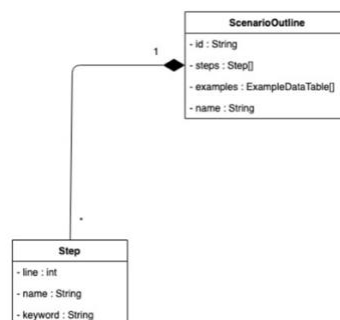


Figure 4.3. UML for Step and Scenario Outline

Apart from the first row, every row in *Examples* data table is executed on *Scenario Outline* as test scenario [5]. So that, *ExampleDataTable* class, which was introduced before, consists of list of *Row* class, that has list of cells as *String*, see also Figure 4.4.

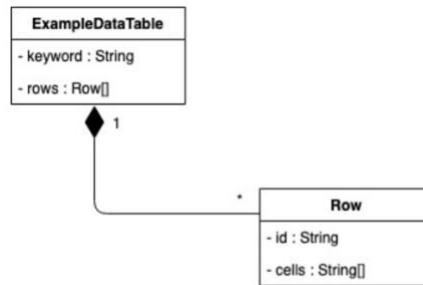


Figure 4.4. UML for Example Data Table and Row

According to Gherkin keyword specifications [5], some of the keywords, which are *Scenario Outline*, *Given*, *When*, *Then*, *And*, *Example*, *Feature* are converted to Java classes, in Figure 4.5, to use following sections. Addition to them, these classes will be called as Gherkin Plain Old Java Object (POJO) in the following parts.

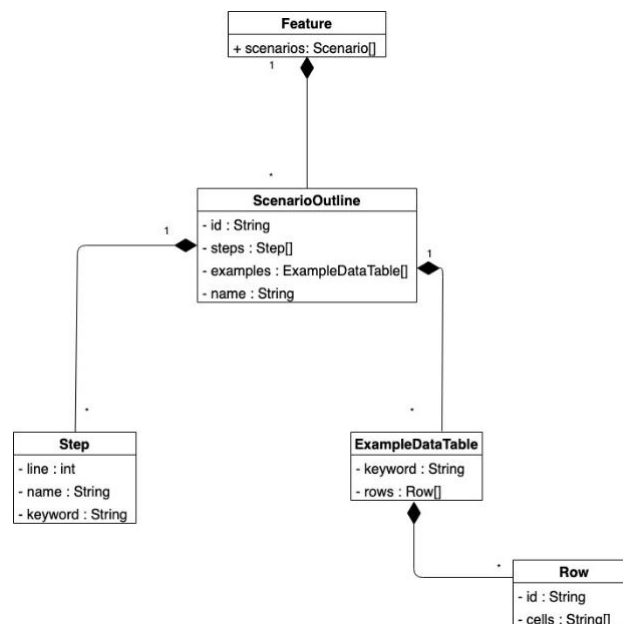


Figure 4.5. UML for All Gherkin Keywords

One of the preliminary tasks to implement the Rule set is writing a parser to parse *Feature* keyword to corresponding Gherkin POJO classes, that are shown in Figure 4.6. A basic class, that is named as *GherkinParser*, is designed with the method, called as *gherkingToPOJO(..)*. It takes only one parameter, that is physical path of the feature file written in SPL-AT Gherkin and returns parsed *Feature* class. So that, it has one dependency to the *Feature* class.

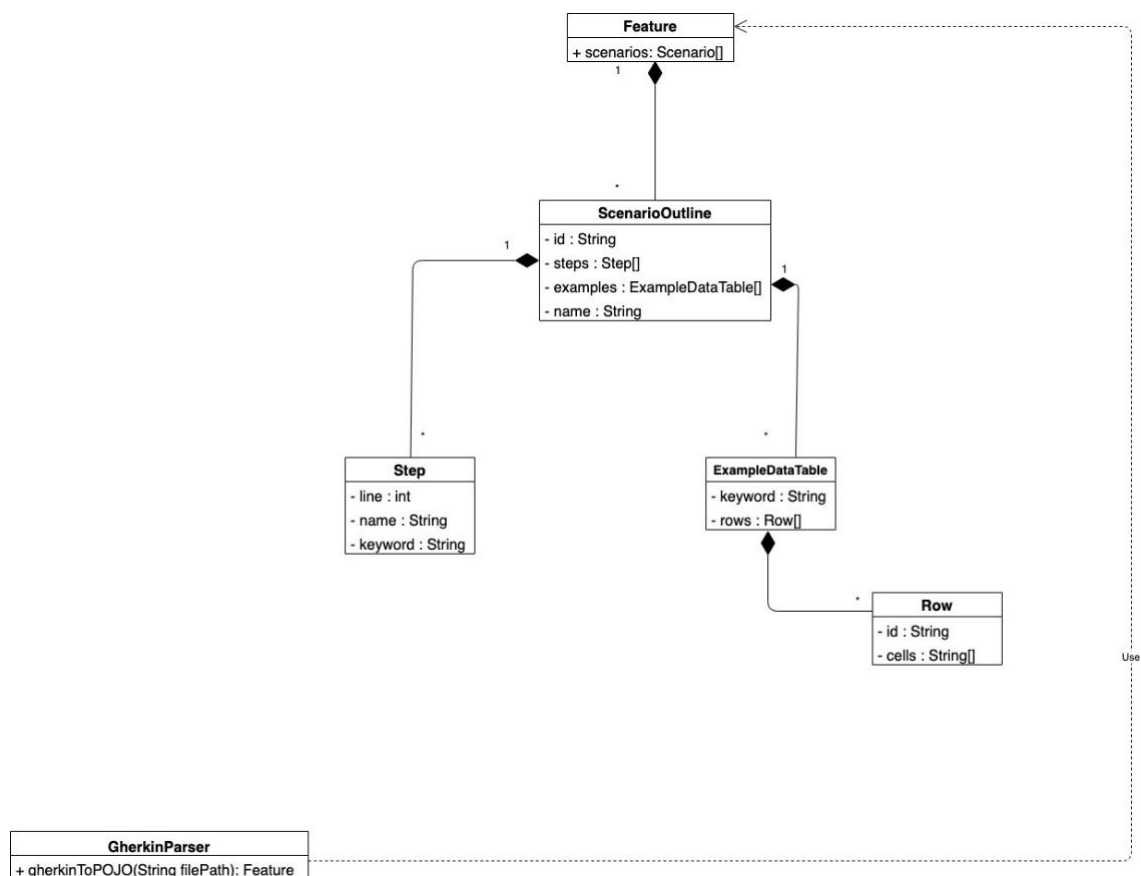


Figure 4.6. UML with Gherkin Parser

Another task is finding the proposed tag structures, which are *address sign* (@) and *dollar sign* (\$), in Chapter 3. Considering consequences of different implementation, an interface named as *ITagFinder* is designed with two methods, which are *findAddressSignTag(...)* and *findDollarSignTag(...)*. Both of them take two parameters as list of *Step* and *keyword* as String. In other words, these methods find the given keyword, such as *PAGE*, and returns the identifier of it.

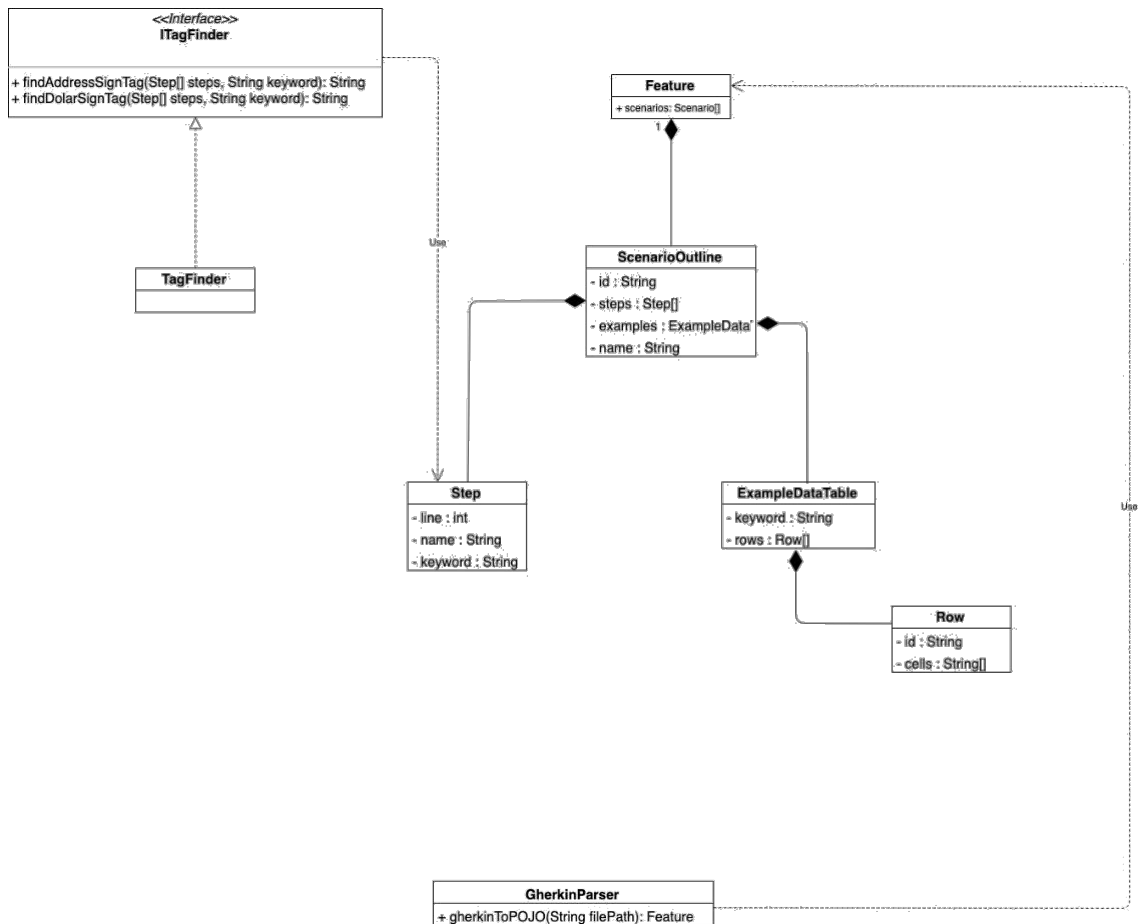


Figure 4.7. UML with Tag Finder

The most complex part is Generator concept, in Figure 4.8, since all business logics that related with rule set are handled in this concept. As first step, *FileGenerator* abstract class is designed to handle common operations such as writing a content to any file respect to file path or writing content of the file to console. And also, as child of the *FileGenerator* class, two different file generators, which are *ClassFileGenerator* and *XMLFileGenerator* abstract classes, are designed. In *ClassFileGenerator* class, there are helper methods to create java methods or inject some line of codes to content of the given method. And also, it has a method to inject some annotations [12] to content of given method content as parameter. It also has two children classes which are *BasePageClassGenerator* and *TestNGClassGenerator*. In a few words, *BasePageClassGenerator* and *TestNGClassGenerator* are responsible for first part of the rule set, which is from Rule 1 to Rule 3, and second part of the rule set, which is from Rule 3 to Rule 9, respectively. On the other hand, *XMLFileGenerator* is less

complex than *ClassFileGenerator*. Because it has only one child class, which is named as *TestNGXMLFileGenerator*. The child class is only responsible for *testng.xml* file, which is focused on Rule 8.

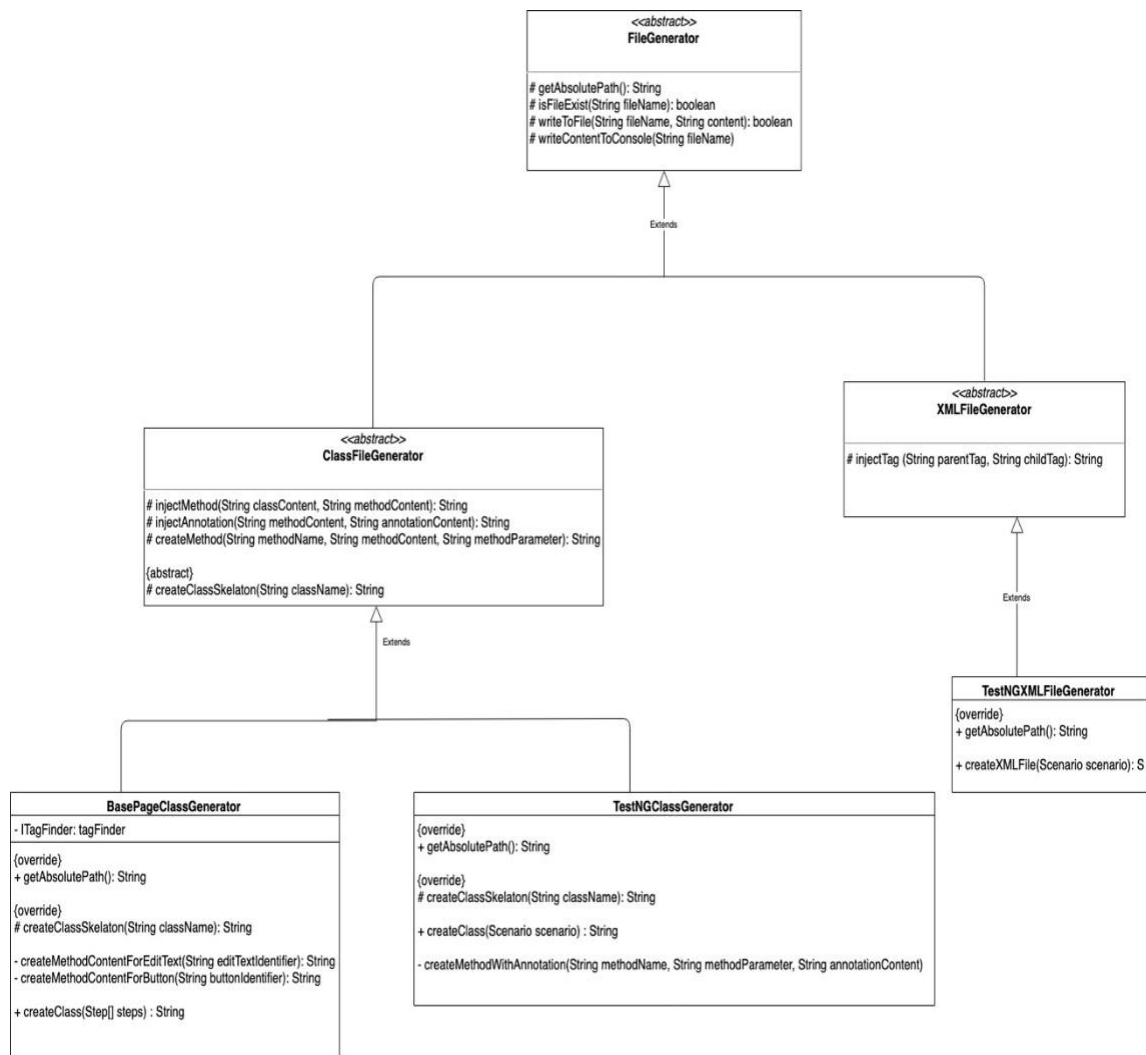


Figure 4.8. UML with Generators

Java Console Application, in Figure 4.9, with *Main* class is generated to execute these concepts on console application, for now. However, these concepts could be improved with web page or any Graphical User Interface framework as future work.

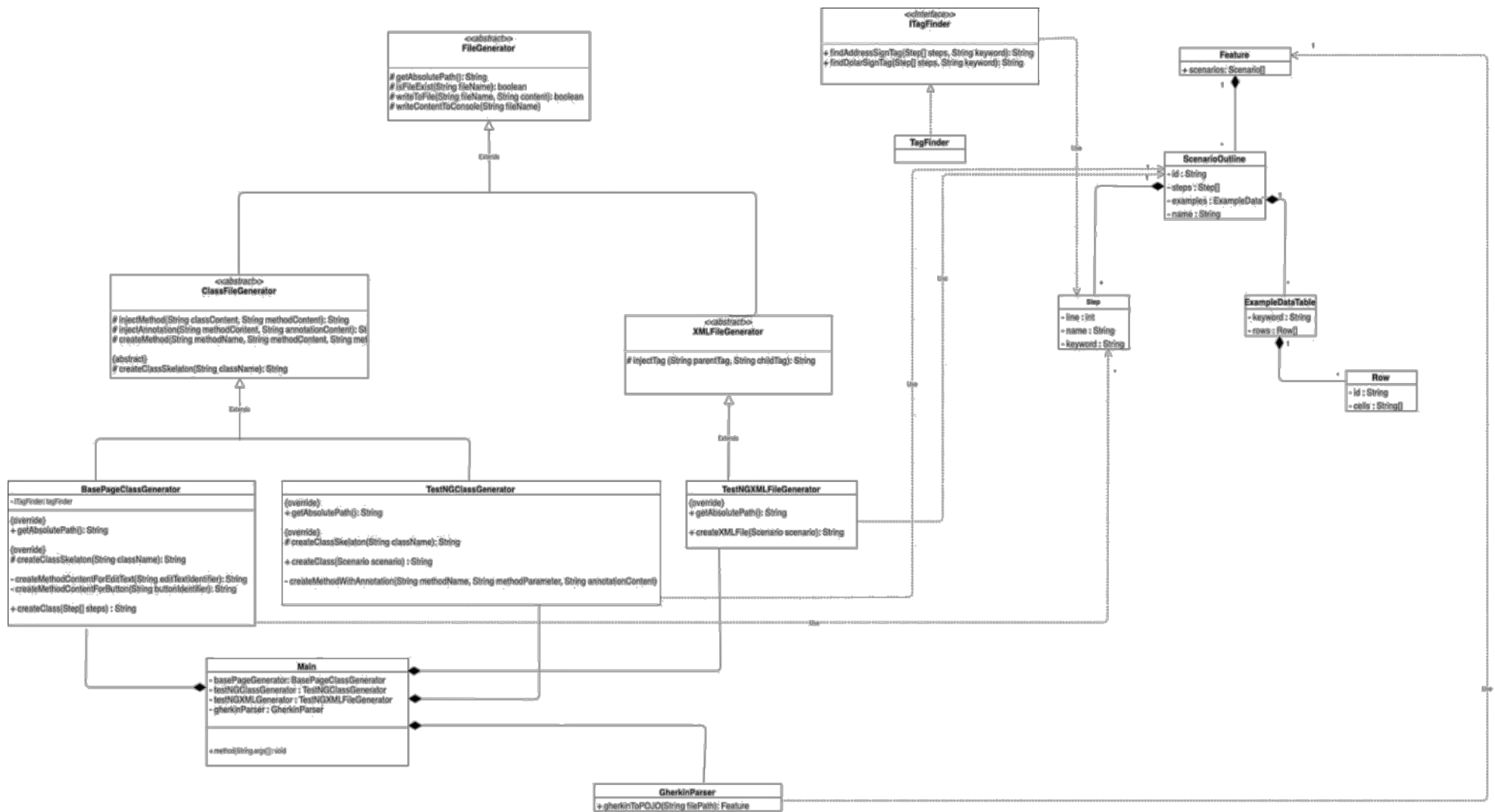


Figure 4.9. UML for Implementation of The Mapping Rule Set

CHAPTER 5

TOOL SUPPORT

Some setups are needed to run implementation of the proposed test method generation technique. In this chapter, these tools and libraries are going to be explained step by step. End of the chapter, anyone who is interested in proposed technique can understand usage of the technique on any local computer.

5.1. Eclipse

Eclipse (<https://www.eclipse.org>) is one of the popular Integrated Development Environment (IDE) for Java developers. During the study, Eclipse IDE for Java Developers Neon 3 Release 4.6.3 is used and anyone can download it from <https://www.eclipse.org/downloads> address.

5.2. Appium

Appium's (<http://appium.io>) desktop application is needed to execute Java client project on any real device or emulators. During the study, Appium desktop application for Mac Version 1.12.1 is used. It also supports Linux and Windows operating systems. Everyone can download it from <http://appium.io/downloads.html> address under *Appium Desktop Apps* title.

5.3. Creating Maven Project

Apache Maven (<https://www.maven.apache.org>) is tool to build and manage Java projects. After downloading Eclipse IDE, an empty maven project could be created under File > New > Project path (Figure 5.3).

5.4. Adding Libraries to Maven Project

If maven project is created successfully, Project Object Model (POM) xml file is created under root directory of the project. In pom.xml file, any library could be added easily with *dependency* tag as child of the *dependencies* tag. Six libraries have to be added on the project which are, testng, appium-java-client, cucumber-core, cucumber-java, cucumber-testng and gson.

5.5. Adding Implementation of The Mapping Rule Set to Maven Project

After adding required libraries, implementation of the mapping rule set, which is explained in Chapter 4 with UMLs, could be added as hardcoded class by class to the maven project. The implementation is developed in Java language with twenty-three classes. All classes are going to be available in *GitHub* repository. Final skeleton of the project is shown in Figure 5.1. In *File* package, java and xml file generators, which are explained in Figure 4.8, are located. In *GherkinReader.GherkinPOJO* package, each concrete component, which are described in Figure 4.5, in Gherkin is implemented. And also, in *Utils* package, some configuration classes with static variables such as describing *@PAGE*, *\$CLICKED* string values in *TagConfiguration.java* or connection strings for mobile device under test in *MobileConfiguration.java* are developed. Additionally, *Feature* directory also should be created in the project because *Scenario Outlines*, which are written in SPL-AT Gherkin, is going to be stored in that location. Also, *MobileConfiguration.java* class should be changed based on application and mobile device, which are going to be tested. When *MainProgram.java* class is executed, children of the *BasePage* and *BaseTestNG* classes are going to be generated automatically in *Pages* and *Tests* directories respectively. Please do not forget refreshing the project after executing *MainProgram.java* class.

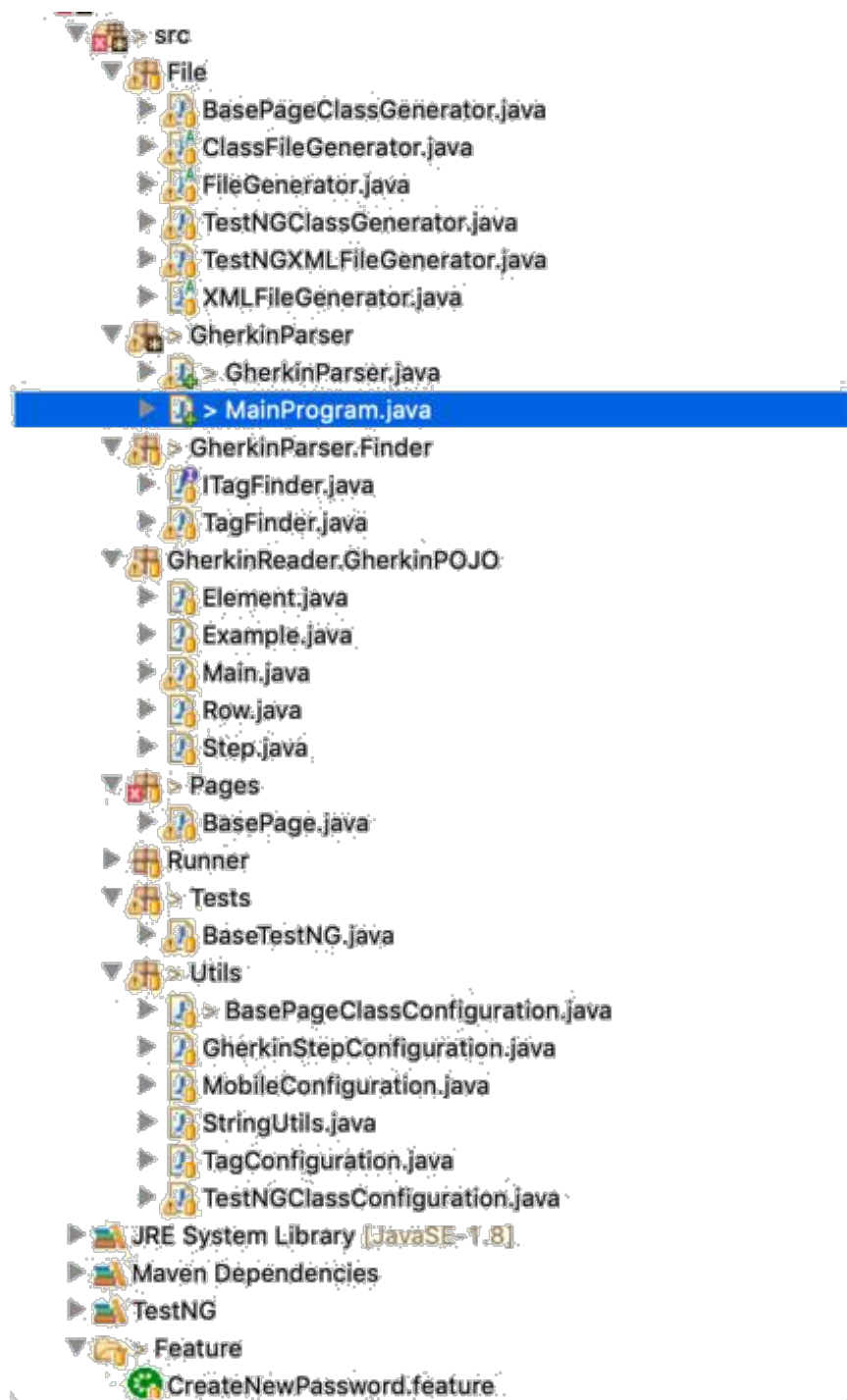


Figure 5.1. Complete Project Structure

CHAPTER 6

CASE STUDY

KidsBusÖ system has five different mobile applications which are KidsBusÖ School Manager, KidsBusÖ School Security, KidsBusÖ Hostess, KidsBusÖ School Staff and KidsBusÖ Parent. And also, these mobile applications have different features in terms of their assignments. For instance, while any user from KidsBusÖ Hostess is responsible for students, who are using his or her bus, users from KidsBusÖ School Security could only deliver students, who will be picked up by trusted parent.

Figure diagrams [13] are tool to represent the feature options in SPLs for user selection. Figure 6.1 is given as an example for feature diagrams. It is a SPL for KidsBus ä, that is chosen as case study in proposed study. KidsBusä is a platform that is developing by Delta Smart Technologies Inc. (www.deltasmart.tech). It provides different types of mobile applications, which are Parent, School Admin, School Security, Hostess and Bus Company, to manage school bus transportation effectively. The root of feature diagram represents the SPL and the nodes are features, which can be mandatory or optional, represented by filled circle and empty circle respectively. Product diagrams, similar to feature diagrams, are user-centric representations of product feature configurations, where all feature selections are made for the product. In Figure 6.1, an example product diagram shows selected features of the product, that is called Gold KidsBusä. Filled and empty circles are removed because the feature selections are completed.

The proposed feature-oriented testing approach with SPL-AT Gherkin provides automatically acceptance tests generation with respect to selected feature combination in product diagram. Analysts and testers could write scenario for customized product in SPL-AT Gherkin, which has a tag structure to refers concrete objects, to generate test methods. The proposed approach follows agile practices for developing software product lines proposed by de Souza and Vilain [14].

Implementation of the mapping rules, which was explained in Chapter 4, will be applied on a mobile application, which is called as KidsBusÖ School Security. School

securities, in KidsBusÖ environment, can display the list of students whom will be taken from the school by an adult. While running the implementation of the mapping rules on the mobile application, five different pages, which are *getting SMS code*, *verifying the SMS code*, *creating new password*, *main* and *login* pages will be tested.

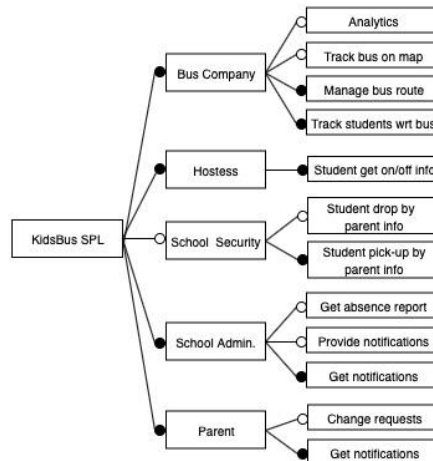


Figure 6.1. SPL Feature Diagram for KidsBusä

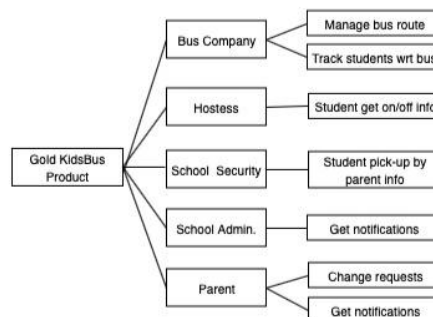


Figure 6.2. SPL Product Diagram for Gold KidsBusä

In the first page, getting SMS code shown in Figure 6.3, there are two different user interface components which are button and edit text. Users can enter their cellphones on edit text and can send the cellphone to KidsBusÖ system with the button. If the cellphone number exists in KidsBusÖ system as school security role, SMS which has verification code will be send the cellphone. Otherwise, the application remains the same page with an error message. Two different test scenarios will be executed on this page. In the first scenario, cellphone number, which belongs to any school security role,

will be executed and then assert that the page is changed or not. On the other hand, in the second scenario, cellphone number, does not belong to any school security role and expected that current page will not be changed. When the scenario outline, shown in Figure 6.4, is executed with the implementation of the mapping rules, mobile application test codes will be generated automatically and run on the mobile application, which is installed on any mobile device. Addition to them, the page also has a *TextView* component shown as “*KidsBus School Security*” in Figure 6.3. Another test scenario, which is shown in Figure 6.5, will be written to be ensure that the Title is shown or not.

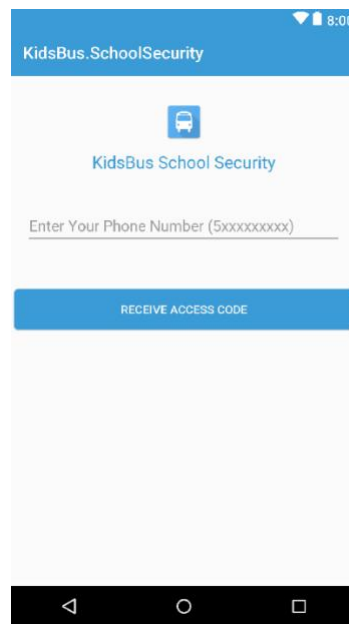


Figure 6.3. Screen for Page Getting SMS Code

Feature: Getting SMS Code

Scenario Outline: Getting SMS code scenario

Given @PAGE ReceiveVerificationCodeActivity is \$OPENED
When <username> is \$ENTERED on @EDIT_TEXT
 usernameInput **And** @BUTTON loginButton is \$CLICKED
Then @PAGE is \$OPENED <page>.

Examples:

username	page
“5454339401”	“.Activity.CommitVerificationCodeActivity”
“5359144691”	“.Activity.ReceiveVerificationCodeActivity”

Figure 6.4. Scenario for Getting SMS Code

Feature: SMS Code Title

Scenario Outline: SMS Code Title scenario

Given @PAGE ReceiveVerificationCodeActivity is \$OPENED
Then @TEXT_VIEW receive_access_activity_app_label
is \$SHOWN

Figure 6.5. Scenario for Getting SMS Code Title

The second page or *verifying the SMS code* page, shown in Figure 6.7, has five different user interface components which are button, edit text and three different text views, like *getting SMS code* page. Users, who has school security role in KidsBusÖ system, should enter the verification code, which is send via SMS to the cellphone, to create user password in the third page, which is called as creating new password page. To test this feature, KidsBusÖ system generates same verification code for all test users. So that, mobile application test project does not need to read content of the SMS. In other words, mobile application test project assumes that verification code is 112233, if the cellphone is verified by KidsBusÖ system as school security role. Two different test scenarios, in Figure 6.8, will be executed as valid and invalid verification code. These scenarios could be extended with different verification code combinations as included character etc. because the code should be formed with numbers. Addition to these two scenarios, three different scenarios for text views, which are *title of the page*, *send again* and *timer for passcode*, will be generated to ensure visibility of them in Figures 6.6, 6.9 and 6.10.

Feature: Verify SMS Code Send Again Text

Scenario Outline: Verify SMS Code Send Again Text scenario Given @PAGE ReceiveVerificationCodeActivity is \$OPENED When <username> is \$ENTERED on @EDIT_TEXT usernameInput And @BUTTON loginButton is \$CLICKED And @PAGE CommitVerificationCodeActivity is \$OPENED Then @TEXT_VIEW send_again is \$SHOWN **Examples:**

```
| username |  
| "5454339401" |
```

Figure 6.6. Scenario for Send Again Text

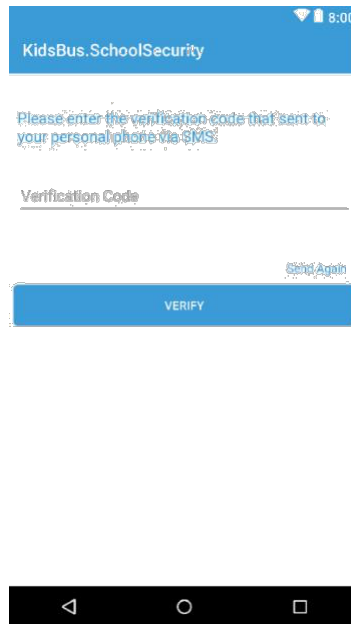


Figure 6.7. Screen for Verify SMS Code

Feature: Verify SMS Code

Scenario Outline: Verify SMS Code scenario

Given @PAGE ReceiveVerificationCodeActivity is \$OPENED
 When <username> is \$ENTERED on @EDIT_TEXT
 usernameInput And @BUTTON loginButton is \$CLICKED
 Then @PAGE CommitVerificationCodeActivity is \$OPENED And
 <passcode> is \$ENTERED on @EDIT_TEXT activation_code
 And @BUTTON loginButton is \$CLICKED again
 And @PAGE is \$OPENED <second_page>

Examples:

username	passcode	second_page
"5454339401"	"111111"	".Activity.CommitVerificationCodeActivity"
"5454339401"	"112233"	".Activity.CreateNewPasswordActivity"

Figure 6.8. Scenario for Verify SMS Code

Feature: Verify SMS Code Count Down Timer Text
Scenario Outline: Verify SMS Code Count Down Timer Text scenario
 Given @PAGE ReceiveVerificationCodeActivity is \$OPENED
 When <username> is \$ENTERED on @EDIT_TEXT
 usernameInput And @BUTTON loginButton is \$CLICKED
 And @PAGE CommitVerificationCodeActivity is \$OPENED
 Then @TEXT_VIEW countdown_timer is \$SHOWN
Examples:
 | username |
 | "5454339401" |

Figure 6.9. Scenario for Count Down Timer Text

Feature: Verify SMS Code Title
Scenario Outline: Verify SMS Code Title scenario
 Given @PAGE ReceiveVerificationCodeActivity is \$OPENED
 When <username> is \$ENTERED on @EDIT_TEXT
 usernameInput And @BUTTON loginButton is \$CLICKED
 And @PAGE CommitVerificationCodeActivity is \$OPENED
 Then @TEXT_VIEW entry_approval_info_text is \$SHOWN
Examples:
 | username |
 | "5454339401" |

Figure 6.10. Scenario for Title Text

In the third page or *creating new password* page, shown in Figure 6.11, users, who are in school security role in in KidsBusÖ system, could create new password with two different edit text and one button user interface components. The critic requirement for this page is that user should enter same password into the these edit text components. Because, KidsBusÖ system should be ensured that given password is confirmed by the user. The test scenario outline, shown in Figure 6.12, is created to test this feature with two different test scenarios as password confirmed and not. Another test scenario could be added as content of the passwords such as strong, weak or non-digit password. And also, the page has an information text view, which starting with “*KidsBus School Security is an application...*” sentence in Figure 6.11. An additional scenario also could be developed for this UI component shown in Figure 6.13.

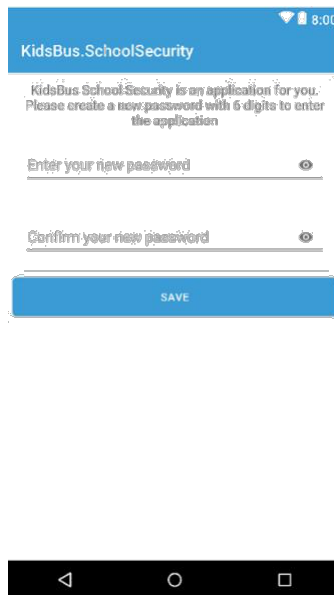


Figure 6.11. Screen for Create New Password

Feature: Create New Password

Scenario Outline: Create new password scenario

Given @PAGE ReceiveVerificationCodeActivity is \$OPENED
 And <username> is \$ENTERED on @EDIT_TEXT
 usernameInput And @BUTTON loginButton is \$CLICKED
 And @PAGE CommitVerificationCodeActivity is \$OPENED
 And <passcode> is \$ENTERED on @EDIT_TEXT
 activation_code And @BUTTON loginButton is \$CLICKED again
 And @PAGE CreateNewPasswordActivity is \$OPENED
 When <new_password> is \$ENTERED on @EDIT_TEXT
 new_password
 And <new_password_confirm> is \$ENTERED on
 @EDIT_TEXT confirm_new_password
 And @BUTTON button_save_new_password is \$CLICKED
 Then @PAGE is \$OPENED <result_page>

Examples:

	username		passcode		new_password		new_password_confirm		
result_page		"5454339401"		"112233"		"555666"		"555555"	
	".Activity.CreateNewPasswordActivity"								

Figure 6.12. Scenario for Create New Password

Feature: Create New Password Title

Scenario Outline: Create New Password Title scenario

Given @PAGE ReceiveVerificationCodeActivity is \$OPENED
And <username> is \$ENTERED on @EDIT_TEXT
usernameInput And @BUTTON loginButton is \$CLICKED
And @PAGE CommitVerificationCodeActivity is \$OPENED
And <passcode> is \$ENTERED on @EDIT_TEXT
activation_code And @BUTTON loginButton is \$CLICKED again
And @PAGE CreateNewPasswordActivity is \$OPENED
Then @TEXT_VIEW titleTextView is \$SHOWN

Examples:

| username | passcode |
| "5454339401" | "112233" |

Figure 6.13. Scenario for Create New Password Title

After user, who has *School Security* role, is introduce himself or herself to the application, main page, which indicates list of students whom will be taken from the school by an adult, is opened shown in Figure 6.14. The page includes one text view on top of itself to show user's name and surname. A scenario could be written to test this text view is shown or not, see also Figure 6.15.

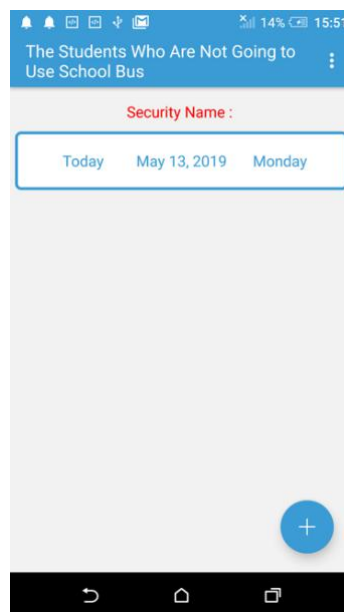


Figure 6.14. Main Screen

Feature: Main User Info Title

Scenario Outline: Main User Info Title scenario

Given @PAGE MainActivity is \$OPENED

Then @TEXT_VIEW main_activity_user_info_label is \$SHOWN

Figure 6.15. Scenario for Main Title

Any user, who is identified by KidsBusÖ, can log out from the application. Afterwards, in any time, user can log in to the application with credentials which are defined on *create new password* page. *Login* page totally includes five different UI components which are welcome text view, username edit text, password edit text, login button and forgot password text view, see also Figure 6.16. One scenario with two different cases could be written to test integrity of the credentials as valid and invalid, see also Figure 6.17. Addition to this test scenario, two different scenarios which are related with visibility of the text views also could be generated, see also Figure 6.18 and Figure 6.19.

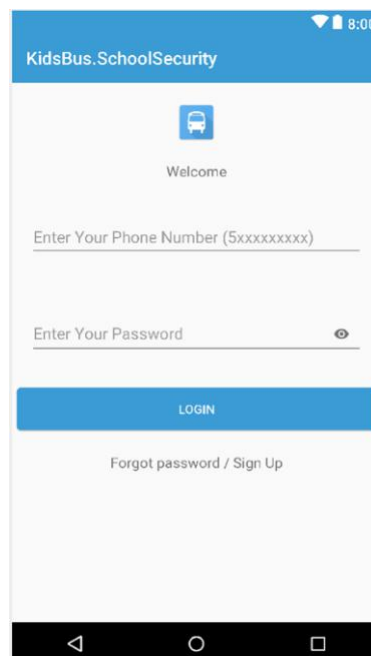


Figure 6.16. Login Screen

Feature: Login

Scenario Outline: Login scenario

```
Given @PAGE MainActivity is $OPENED
And @BUTTON logout is $CLICKED
And @PAGE LoginActivity is $OPENED
When <username> is $ENTERED on @EDIT_TEXT
usernameInput And <password> is $ENTERED on @EDIT_TEXT
passwordInput And @BUTTON loginButton is $CLICKED
Then @PAGE is $OPENED <second_page>.
```

Examples:

```
| username | password | second_page |
| "5454339401" | "123456" | ".Activity.MainActivity" |
| "5454339401" | "111111" | ".Activity.LoginActivity" |
```

Figure 6.17. Scenario for Credential Integrity

Feature: Login Forgot Password Title

Scenario Outline: Login Forgot Password Title scenario

```
Given @PAGE MainActivity is $OPENED
And @BUTTON logout is $CLICKED
And @PAGE LoginActivity is $OPENED
Then @TEXT_VIEW forgottenPassword is $SHOWN
```

Figure 6.18. Scenario for Forgot Password Title

Feature: Login Welcome Title

Scenario Outline: Login Welcome Title scenario

```
Given @PAGE MainActivity is $OPENED
And @BUTTON logout is $CLICKED
And @PAGE LoginActivity is $OPENED
Then @TEXT_VIEW welcome is $SHOWN
```

Figure 6.19. Scenario for Welcome Title

In Chapter 6, five different pages in KidsBus[®] School Security mobile application, which are *getting SMS code*, *verifying the SMS code*, *creating new password*, *main* and *login*, are tested with the twelve different feature files, that is written in SPL-AT Gherkin. While five different children of the *BasePage* (see also Chapter 3.1) classes are created with the mapping rules, and also, twelve different children of the *BaseTestNG* (see also Chapter 3.2,) classes are generated. Fifteen different test case scenarios are also covered with generated mobile application test project.

Test reports of these scenarios are represented step by step as *fail* or *passed* by eclipse console for TestNG framework, see also Figure 6.20. In other words, any particular study for Test Reporting is not covered in scope of the thesis. Custom test report generation also is easy to handle with Listeners and Reporters, which implement `org.testng.ITestListener` and `org.testng.IReporter` interfaces respectively, in TestNG framework. Addition to them, while the report could be generated as PDF format which is observed end of test scenarios execution, it also monitored as real-time with Graphical User Interface (GUI) supports such as progress bar etc. [18].

Test scenarios also could be extended with different combination of the *Examples* data table in case study. While the scenarios are increasing, automatic code generation time is also increasing. Generation time in milliseconds for each scenario are demonstrated in Figure 6.2. These values are calculated with `System.currentTimeMillis()` method which returns the current time in milliseconds from January 1, 1970 Universal Time Coordinated (UTC) to current time. Total generation time for implementation of the twelve test scenarios, which are written in SPL-AT Gherkin, is three hundred and thirty-seven milliseconds. In other words, acceptance test project, which covers twelve different test scenarios, for KidsBus School Security mobile application is generated in three hundred and thirty-seven milliseconds without any test framework knowledge.

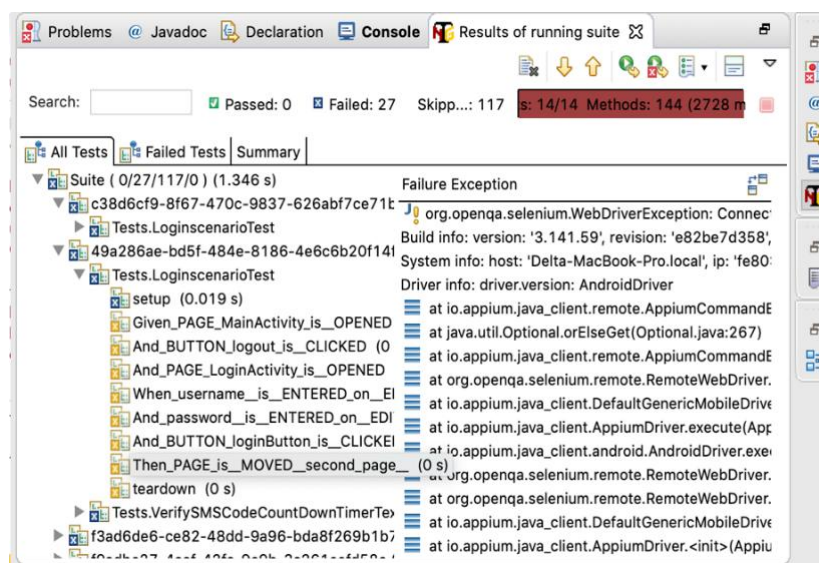


Figure 6.20. Test Results

Execution Order	Scenario Name	Generation Time in Ms
1	Login Scenario	229
2	Verify SMS Code Count Down Timer Text	13
3	Create New Password Title	14
4	Verify SMS Code	12
5	Forgot Password	11
6	Verify SMS Code Title	10
7	Login Welcome Title	7
8	Main User Info Title	7
9	Create New Password	15
10	Getting SMS Code	8
11	SMS Code Title	4
12	Send SMS Code Again	7
-	-	337

Figure 6.21. Scenarios' Generation Time in Milliseconds

CHAPTER 7

RELATED WORKS

7.1. Robot Framework

Robot framework which is open source automation framework hosted on GitHub for acceptance testing, acceptance test driven development, and robotic process automation [16]. It is also released under Apache License 2.0 and anyone can download it from official web page which is robotframework.org. Firstly, it was developed by Nokia Networks (networks.nokia.com). In these days, it is supported by its own foundation which is called as Robot Framework Foundation (robotframework.org/foundation/). Also, it uses keyword-driven testing approach, which is called as table-driven testing or action word-based testing.

When Robot framework and proposed feature-oriented testing approach with SPL-AT Gherkin are compared, not only similarities but also differences are found. And, they are going to be focused in the following paragraphs.

The framework syntax has different special keywords, which are *Settings*, *Variables*, *Keywords* and *Test Cases* to used different purposes, see also Figure 7.1. So that, everyone who is interested in Robot framework has to learn its own syntax firstly. On the other hand, in proposed approach, everyone, who has experience about Gherkin, can adopt SPL-AT Gherkin in a short time with cost of learning tags which are *addressing (@)* and *dollars (\$)* signs. While test data and identifier of any user interface component such as Button could be written in ***Variable*** section as weak practice in robot framework, this differentiation is already handled in SPL-AT Gherkin with *addressing sign (@)* and *Examples* data table Figure 7.2 and Figure 7.3 compare robot framework with SPL-AT Gherkin.

There are also similarities between Robot Framework and proposed feature-oriented testing approach with SPL-AT Gherkin. Major similarity is that both of them are working on acceptance testing and acceptance test driven development and generate the test project automatically based on their syntax rules. And also, they hide technical details with spoken language keywords such as ***Test Cases***, *Click Element*,

@*EDIT_TEXT* and *ENTERED* to be clear for project team that includes not only technical but also non-technical members.

```
*** Settings ***
...     Settings here.

*** Variables ***
...     Variables here.

*** Test Cases ***
...     Test Cases here.

*** Keywords ***
...     Keywords here.
```

Figure 7.1. Robot File with Empty Skeleton

```
*** Settings ***
Library AppiumLibrary
....

*** Variables ***
${BTN_ID} = id=this_is_button_identifier
${EDITTEXT_ID} = id=this_is_edit_text_identifier
${CONTENT} = this_is_content
....

*** Test Cases ***
Add Content And Submit Button Is Clicked
    Add Content ${CONTENT}
    Submit Button
    ....

*** Keywords ***
Submit Button
    Click Element  ${BTN_ID}

Add Content
    [Arguments] ${content}
    Input Text ${EDITTEXT_ID} ${content}
    ....
```

Figure 7.2. Sample Robot with Appium

```

Feature: This is the title of the Feature

Scenario Outline: This is the title of the Scenario Outline
Given ...
When <delimited_parameter> is $ENTERED
    on @EDIT_TEXT this_is_edit_text_identifier
And @BUTTON this_is_button_identifier is $CLICKED
Then ...

Examples:
| delimited_parameter |
| "this_is_content" |

```

Figure 7.3. Sample SPL-AT Gherkin

7.2. Cucumber

Cucumber is a tool that scans executable specifications, which are written in plain text, and validates the software which is responsible for those specifications [17]. It is also based on Behavior Driven Development, that is briefly explained in Chapter 2, to write acceptance tests. Cucumber indicates that the software is success or failure based on each scenario. It has also MIT License and everyone can follow and download it from its official GitHub page (<https://github.com/cucumber>). Each scenario in Cucumber is written in Gherkin, that is also explained in Chapter 2, therefore, everyone, who wants to use Cucumber tool, should learn Gherkin syntax rules.

The main similarity between Cucumber tool and proposed feature-oriented testing approach with SPL-AT Gherkin is language of the acceptance scenarios. In other words, anyone, who worked before with Cucumber, can adapt SPL-AT Gherkin in a short time. Another similarity is that both of them generate acceptance test methods and classes automatically based on their domain specific languages.

While proposed feature-oriented testing approach with SPL-AT Gherkin is automatically implement the generated acceptance test methods and classes, Cucumber only generates skeleton without any implementation. For instance, when sample scenario in Figure 7.4, which is written in Gherkin, is run, Figure 7.5 is generated automatically by Cucumber.

Feature: Is it Friday yet?
Everybody wants to know when it's Friday

Scenario: Sunday isn't Friday
Given today is Sunday
When I ask whether it's Friday yet
Then I should be told "Nope"

Figure 7.4. Sample Scenario in Gherkin for Cucumber

```
@Given("^today is Sunday$")
public void today_is_Sunday() {
// Write code here that turns the phrase above into //concrete
actions throw new PendingException();
}
@When("^I ask whether it's Friday yet$")
public void i_ask_whether_it_s_Friday_yet() {
// Write code here that turns the phrase above into //concrete
actions throw new PendingException();
}
@Then("^I should be told \"([^\"]*)\"$")
public void i_should_be_told(String arg1) {
// Write code here that turns the phrase above into //concrete
actions throw new PendingException();
}
```

Figure 7.5. Generated Test Method by Cucumber

7.3. Gauge

Gauge is open source framework for test automation especially acceptance tests (<https://gauge.org/>). It is also released under GNU Public License version 3.0 (<http://www.gnu.org/licenses/gpl-3.0.txt>) and available its official GitHub page (<https://github.com/getgauge/gauge>). It has own syntax that is not Given-When-Then style, however it is understandable for everyone like Gherkin. It works with different languages such as JavaScript, C#, Java, Python, Ruby.

The common purpose for Gauge and proposed solution is generating acceptance test cases against software which has User Interface components. Gauge works with Taiko (<https://github.com/getgauge/taiko>), which is a free and open source browser

automation tool, to generate test cases. Taiko works on Linux, MacOS and Windows, however, it supports Chrome web browser because it uses the Chrome DevTools API (<https://chromedevtools.github.io/devtools-protocol/tot/Browser>). As an example, the specification file shown in Figure 7.6, which is written in Gauge syntax, generates the step implementation file shown in Figure 7.7, that uses Taiko automation tool.

```
## Search Google

* Goto Google's search page
* Search for "github Taiko"
* Page contains "getgauge/taiko"
```

Figure 7.6. Sample Scenario in Gauge Syntax

```
/* globals gauge*/
"use strict";
const { openBrowser,write, closeBrowser, goto, press,text, contains } =
require('taiko');
const assert = require("assert");
const headless = process.env.headless_chrome.toLowerCase() === 'true';

beforeSuite(async () => {
  await openBrowser({ headless: headless })
});

afterSuite(async () => {
  await closeBrowser();
});

step("Goto Google's search page", async () =>
  { await goto('http://google.com');
});

step("Search for <query>", async (query) => {
  await write(query);
  await press('Enter');
});

step("Page contains <content>", async (content) =>
  { assert.ok(await text(contains(content)).exists());
});
```

Figure 7.7. Implementation File for Gauge

CHAPTER 8

CONCLUSION

In this study, a feature-oriented testing approach is proposed for platform-based SPLs through a novel extension to Gherkin called SPL-AT Gherkin and a novel automatic test method technique based on TestNG framework.

KidsBusÖ system, which is platform that manages the school bus transportation process, is selected as a case study. Five different pages in KidsBusÖ School Security, which are *getting SMS code*, *verifying the SMS code*, *creating new password*, *main* and *login* are tested with twelve different feature files written in SPL-AT Gherkin. These feature files generate five and twelve different children of the *BasePage* and *BaseTestNG* classes respectively. And also, fifteen different test cases are covered without any technical implementation such as writing test suite or method in TestNG framework with these classes. The test cases could be increased with additional feature files that have different combination of test data on *Examples* data table. In terms of test case generation, future task is increasing test cases using reusable feature files. Thanks to more test cases, coverage percentage could be rising. In other words, generating more test cases using less feature file is one of the objectives.

While generating test cases during case study, importance of execution order for complete test scenarios such as *Getting SMS code scenario* (Figure 6.4) and *Main User Info Title scenario* (Figure 6.15) was observed. Because, the user should be authenticated by the KidsBusÖ API to reach main page in KidsBusÖ School Security. In other words, some test scenarios have to be run before the others. To solve this challenge, a test case management tool for acceptance tests that are generated by feature files written in SPL-AT Gherkin is going to be developed as another future work. The tool could be an extension for SPL-AT Gherkin such as another tag structure like address sign (@) or dollar sign (\$) or a platform that has some UI components to be more understandable and administrable by project management team. While developing automatic test method technique that is explained in Chapter 4 *Implementation of The Mapping Rule Set*, the solution is designed based on SOLID principles of object-oriented programming [19], which was introduced by Robert C. Martin, as much as

possible. Addition to it, connection the proposed approach with input contract testing based on Event Sequence Graphs [15] is planned, so that coverage-based test generation can be achieved for platform-based SPLs.

REFERENCES

- [1] Introducing BDD. (2017, February 09). Retrieved April 15, 2019, from <https://dannorth.net/introducing-bdd/#translations>
- [2] Bliki: GivenWhenThen. (n.d.). Retrieved April 15, 2019, from <https://www.martinfowler.com/bliki/GivenWhenThen.html>
- [3] Four-Phase Test. (n.d.). Retrieved April 15, 2019, from <http://xunitpatterns.com/FourPhaseTest.html>
- [4] Bliki: PageObject. (n.d.). Retrieved April 15, 2019, from <https://martinfowler.com/bliki/PageObject.html>
- [5] Gherkin Reference. (n.d.). Retrieved April 15, 2019, from <https://docs.cucumber.io/gherkin/reference/>
- [6] Welcome. (n.d.). Retrieved April 15, 2019, from <https://testng.org/doc/index.html>
- [7] UDID. (2019, February 01). Retrieved April 15, 2019, from <https://en.wikipedia.org/wiki/UDID>
- [8] (n.d.). Retrieved April 15, 2019, from <https://testng.org/doc/documentation-main.html>
- [9] Universally unique identifier. (2019, April 15). Retrieved April 15, 2019, from https://en.wikipedia.org/wiki/Universally_unique_identifier
- [10] Edit this Doc Get Current Activity. (n.d.). Retrieved April 15, 2019, from <http://appium.io/docs/en/commands/device/activity/current-activity/>
- [11] Edit this Doc Is Element Enabled. (n.d.). Retrieved April 15, 2019, from <http://appium.io/docs/en/commands/element/attributes/enabled/>
- [12] Annotations Basics. (n.d.). Retrieved April 15, 2019, from <https://docs.oracle.com/javase/tutorial/java/annotations/basics.html>
- [13] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, “Feature-oriented domain analysis (FODA) feasibility study,” Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1990.
- [14] D. S. de Souza, P. Vilain. “Selecting Agile Practices for Developing Software Product Lines”, International Conference on Software Engineering & Knowledge Engineering (SEKE 2013), 220-225, 2013.
- [15] T. Tuglular, F. Belli, and M. Linschulte. Input contract testing of graphical user interfaces. International Journal of Software Engineering and Knowledge Engineering, 26(02), pp.183-215, 2016.

- [16] Robot Framework. (n.d.). Retrieved May 3, 2019, from <https://robotframework.org/>
- [17] Cucumber. (n.d.). Retrieved May 13, 2019, from <https://cucumber.io/>
- [18] (n.d.). Retrieved June 10, 2019, from <https://testng.org/doc/documentation-main.html#logging>
- [19] Martin, R. C. (2000). Design principles and design patterns. *Object Mentor*, 1(34), 597.